Zsongor F. GOBESZ Ciprian BACOȚIU

INIȚIERE ÎN PROGRAMARE ȘI ÎN LIMBAJUL FORTRAN

U.T.PRES

Zsongor F. GOBESZ **Ciprian BACOȚIU**

INIȚIERE ÎN PROGRAMARE ȘI ÎN LIMBAJUL FORTRAN



Editura U.T. PRES
Cluj-Napoca, 2003

PREFAȚĂ

Această carte se adresează în primul rând studenților Colegiilor de Construcții şi Instalații din cadrul Universității Tehnice din Cluj-Napoca. Ea poate fi utilă însă în egală măsură tuturor celor care doresc să se inițieze în limbajul de programare Fortran.

Conținutul cărții a fost selectat judicios, astfel încât pe lângă caracterul didactic specific să prezinte şi un pronunțat caracter practic aplicativ şi să nu necesite cunoştințe inițiale speciale din partea cititorului, în domeniul utilizării calculatoarelor personale. La elaborarea lucrării s-a avut în vedere accesibilitatea unor instrumente informatice cu licență gratuită şi perspectiva utilizării limbajului Fortran la rezolvarea problemelor tehnico-ştiințifice din domenii diverse. Exemplele sunt numeroase şi sugestive, din cele mai diferite domenii, în general simple, spre a fi accesibile studenților din anii mici de studiu şi vizează, pe lângă fixarea şi adâncirea treptată a cunoştințelor necesare programării, formarea unei gândiri analitice, de alegere şi formulare adecvată a algoritmilor de calcul. Spre deosebire de alte lucrări similare, s-a urmărit ca în cadrul acestui volum să fie prezentate unitar majoritatea aspectelor principale legate de realizarea programelor în limbajul Fortran. Prezentarea teoretică succintă împreună cu exemplele tratate ajută după părerea noastră la fixarea temeinică a cunoştințelor şi fac ca această carte să fie utilă şi acelora care prin autoinstruire doresc să se inițieze în programarea calculatoarelor.

Cele 6 capitole ale cărții tratează următoarele aspecte:

- În capitolul 1 cititorul este inițiat în evoluția sistemelor automate de calcul, a nivelelor de comunicare om-calculator.
- Capitolul 2 este consacrat prezentării rolului şi modului de alcătuire a algoritmilor precum şi a câtorva dintre instrumentele utilizate pentru descrierea acestora.
- În capitolul 3 se prezintă sintaxa şi semantica limbajului Fortran 77 într-o formă sintetică, considerată utilă pentru realizarea aplicațiilor de tip consolă.
- Capitolul 4 tratează diferențele specifice ale limbajului Fortran 90 față de Fortran 77, punctând facilitățile şi avantajele apărute prin evoluția limbajului, însă fără pretenția de a constitui o referință completă. Multe dintre aceste noi caracteristici sunt ilustrate prin exemple simple şi sugestive, din considerente practice.
- Capitolul 5 este destinat prezentării sumare a compilatorului GNU Fortran 77 (g77), cel mai cunoscut şi mai accesibil compilator la ora actuală.
- Capitolul 6 conține exerciții (exemple complete şi comentate) de dificultăți gradate. Soluțiile propuse au fost testate cu ajutorul compilatorului g77, din acest motiv conțin şi câteva facilități noi întâlnite în Fortran 90 şi acceptate de acest compilator.

În încheiere, dorim să mulțumim atât membrilor de familie cât şi prietenilor, colegilor de la Facultatea de Construcții care ne-au susținut şi sprijinit în elaborarea acestei lucrări.

Aprilie 2003,
Cluj-Napoca

Autorii

CUPRINS

CAPITOLUL 1: INTRODUCERE

1.1 NOȚIUNI DESPRE CALCULATOARE ŞI PRELUCRAREA DATELOR

Oamenii au fost fascinați probabil dintotdeauna de capacitatea de lucru a maşinilor. Este evident că din această cauză există o dorință profundă pentru crearea acestora. Dacă privim înapoi în istoria tehnicii, de la mecanismele vechi (cum ar fi scripetele, cântarul etc.) până la sistemele cele mai noi (cum ar fi digitizoarele, echipamentele de comunicații etc.) întotdeauna au existat încercări pentru copierea şi reproducerea unor soluții naturale prin mijloace artificiale. Un farmec aparte caracterizează tentativele de imitare ale unor capacități umane sau realizarea unor instrumente pentru extinderea acestor capacități (de exemplu celebrul şi misteriosul jucător automat de şah, realizat de Kempelen). Versiunile moderne ale acestora se referă la roboți, androizi şi alte caracteristici ştiințifico-fantastice (de la Frankenstein la Star Trek). În planul ştiințelor teoretice, filozofii din Grecia antică au propus deja crearea unor mecanisme formale bazate pe logică. Câteva variante, realizate în cursul timpului, au la bază construcția unor modele de raționare mecanizată. Un exemplu în acest sens ar putea fi maşina construită de Ramon Lull din Spania, pentru demonstrarea existenței lui Dumnezeu. Lull a folosit caractere ca simboluri pentru reprezentarea cuvintelor (şi a argumentelor), precum şi combinații ale acestora pe baza unui sistem de reguli. După cum reiese din descrieri, la baza acestui sistem a stat o schemă mecanică prin care era posibilă realizarea unei varietăți de figuri geometrice care, dacă erau mişcate unele față de altele, determinau noi argumente logice. Sistemul conceput de Lull a fost limitat de stadiul de dezvoltare al geometriei tridimensionale prin numărul operațiilor geometrice posibile. Asemenea invenții rămâneau de multe ori secrete sau ajungeau să fie privite ca nişte jucării interesante.

O maşină ce poate manipula simboluri este şi calculatorul. Principala virtute a acestuia este viteza de operare (numărul ridicat de operații realizate într-un interval scurt de timp). Utilizarea eficientă a acestor echipamente este posibilă prin programarea lor, transpunând astfel gândirea omului în aceste maşini. Folosirea sistemelor de calcul s-a extins astfel de la aplicațiile contabile, financiar-bancare, până la aplicațiile inginereşti, de la recunoaşterea şi sinteza sunetelor până la modelarea virtuală. Electronica şi informatica, tehnicile de calcul şi automatizările, sistemele de comunicații sunt domenii care fac parte din viața noastră cotidiană. Mai mult chiar, dezvoltarea acestor domenii a devenit o necesitate pentru modul în care se măsoară azi progresul societății umane. La actualul grad de dezvoltare al ştiinței şi tehnicii volumul informațiilor a crescut foarte mult şi continuă să crească în ritm accelerat. Cantitatea de informații ce intervine în caracterizarea unui fenomen depinde pe de o parte de complexitatea acestuia, iar pe de altă parte de profunzimea cu care el trebuie cunoscut. În marea lor majoritate fenomenele sunt complexe şi se urmăreşte descrierea lor cât mai exactă. Pentru a putea manevra informațiile, acestea trebuie modelate. *Informația* este constituită prin juxtapunerea de simboluri grupate convențional pentru a reprezenta evenimente, obiecte, idei şi relații între aceste diverse elemente. Modelul manevrabil al informațiilor considerate elementare poartă denumirea generică de *date*. Conform celor enunțate mai sus, este necesar deci să se prelucreze un volum mare de date.

Pentru unele procese apar în plus condiții legate de precizia calculelor. Ca atare, de cele mai multe ori trebuie să se lucreze cu multe cifre semnificative, volumul calculelor crescând astfel. De asemenea, timpul afectat rezolvării problemelor, oricât de complexe ar fi acestea, este limitat. Toate acestea atestă utilitatea folosirii calculatoarelor care concomitent cu viteza mare de calcul pot asigura şi precizii de calcul care satisfac exigențele, oferind o productivitate mărită. Totodată ele preiau o însemnată parte din eforturile intelectuale necesitate de efectuarea calculelor, ceea ce permite concentrarea acestor eforturi asupra muncii de creație. Paradoxal însă, prin volumul mare de calcule creşte şi efortul necesar stăpânirii problemelor abordate şi rezolvate, prin interpretarea corespunzătoare a volumului crescut de rezultate. Deci în prezent, direct sau indirect, mijloacele moderne de calcul contribuie din plin la orice realizare a ştiinței şi tehnicii, în tot mai multe sectoare de activitate devenind nerentabilă izolarea de calculator.

Cele de mai sus constituie o explicație a avântului extraordinar pe care informatica l-a înregistrat în ultimele decenii. Prin *informatică* (neologism creat în 1962 prin alăturarea şi juxtapunerea parțială a cuvintelor *infor*mație şi auto*matică*) se înțelege în general tehnica prelucrării automate şi raționale a informației, suportul cunoştințelor şi al modului de comunicare uman. Având în vedere cantitatea şi complexitatea informațiilor, putem afirma că mnemonica *informatica* acoperă o arie largă a tehnicilor şi metodologiilor legate de punerea în funcțiune a dispozitivelor complexe reprezentate de calculatoare şi sisteme informatice. Informatica se ocupă atât de natura informației (care-i serveşte drept materie primă) cât şi de metodele care permit tratarea şi prelucrarea lor, precum şi de mijloacele care pot fi puse în funcțiune pentru efectuarea concretă a acestei prelucrări. Domeniile de aplicare ale informaticii se regăsesc în toate sferele de activitate ale lumii contemporane.

La culegerea şi prelucrarea automată a datelor, un rol deosebit este jucat de erorile întâlnite. Chiar dacă acestea nu pot fi eliminate în totalitate, recunoaşterea, stăpânirea şi limitarea acestora are un rol important din perspectiva rezultatelor urmărite. Există trei categorii de erori ce nu pot fi eliminate, din acest motiv necesită o atenție deosebită pe parcursul tratării datelor:

1. *Erorile inerente* care țin de instrumentele de măsură utilizate la achiziționarea datelor. Aceste instrumente dispun de anumite caracteristici legate de natura, alcătuirea şi funcționarea lor. Indiferent dacă este vorba de un liniar simplu sau de un instrument optic de mare precizie, va exista o eroare la citirea datelor măsurate, din cauza grosimii fizice a gradațiilor şi ale reperelor utilizate.

2. *Erorile de metodă* se datorează modului de selectare a algoritmilor şi procedeelor de prelucrare. Pentru aceeaşi problemă se pot alege mai multe abordări, mai multe metode de rezolvare. Unele dintre aceste metode pot fi mai exacte decât altele, însă aplicarea unor metode indirecte va conduce inevitabil la considerarea unor toleranțe în funcție de raportul de rentabilitatate generat de costuri şi rezultate.

3. *Erorile de calcul* se nasc din modul de reprezentare valorică a datelor şi rezultatelor. Spațiul fizic utilizat ca memorie pentru reprezentarea valorilor numerice fiind limitat, vor apare inevitabil trunchieri şi rotunjiri.

Din punctul de vedere al clasificării calculatoarelor putem vorbi de trei clase mari:

1. *Calculatoarele numerice* (cifrice sau digitale) prelucrează cantități sau mărimi discrete reprezentate prin valori cu un număr finit de cifre de reprezentare semnificative. Avantajele principale ale acestor calculatoare constau în universalitatea utilizării, precizia ridicată a soluțiilor şi adaptabilitatea structurii grație modulării (în funcție de complexitatea problemei de rezolvat). Calculatoarele personale fac parte din această categorie.
2. *Calculatoarele analogice* operează cu mărimi ce pot varia continuu. Aceste calculatoare au un domeniu mai limitat de aplicare (din motive tehnologice) şi se folosesc mai ales la rezolvarea unor probleme fizice, care din punct de vedere matematic se pot modela prin sisteme de ecuații diferențiale. Precizia soluțiilor furnizate de aceste echipamente este limitată de precizia cu care funcționează diferitele componente ale calculatorului. Având în vedere că multe probleme ale mecanicii construcțiilor se pot modela matematic prin sisteme de ecuații liniare (sau diferențiale), au fost realizate în diferite țări şi calculatoare specializate pentru rezolvarea unor asemenea probleme, dar având un domeniu restrâns de aplicare nu au putut concura calculatoarele numerice universale.
3. *Calculatoarele electronice mixte* (hibride) rezultă de fapt din asocierea celor două clase precedente cumulând avantajele lor.

În cele ce urmează ne vom referi doar la calculatoarele numerice (nespecializate). Apariția şi dezvoltarea calculatoarelor electronice este de un dinamism de-a dreptul exploziv. Pentru a marca din punct de vedere constructiv progresele înregistrate în această ramură a ştiinței şi tehnicii, perioada scursă din anul 1946 (când a apărut ENIAC, primul calculator electronic) şi până în prezent, a fost împărțită în etape, fiecare reprezentând o *generație* de calculatoare.

Calculatoarele din prima generație (1946—1953) aveau următoarele caracteristici principale:

- utilizau tuburi electronice;
- aveau numai memorie internă cu o capacitate redusă;
- pentru introducerea datelor şi extragerea rezultatelor utilizau de regulă bandă perforată;
- efectuau un număr de câteva sute până la câteva mii de operații elementare pe secundă;
- scrierea programelor se făcea numai în cod maşină (sau prin conectică), totuşi conceptele de asamblor şi subprogram sunt deja folosite;
- siguranța în funcționare era redusă (a se vedea şi originea termenului *debugging*: depanare prin eliminarea insectelor atrase de lumina tuburilor electronice – noțiunea a fost consacrată prin însemnările de întreținere ale calculatorului Mark I de la universitatea Harvard).

După 1953 apar calculatoarele din generația a doua, cu următoarele caracteristici:

- tuburile electronice sunt înlocuite cu tranzistori şi se folosesc circuite imprimate;
- în afara memoriei interne (mai extinse decât la generația precedentă) apare şi memoria externă (banda magnetică);
- viteza de operare creşte (sute de mii de operații elementare pe secundă);
- elementele periferice se dezvoltă foarte mult (pentru introducerea informației se utilizează cartele perforate dar şi benzi magnetice, la extragerea rezultatelor se folosesc imprimante rapide),
- apar limbajele de programare de nivel ridicat, asociate noțiunii de macro-asamblor;
- apar noțiunile de *hardware* (ansamblul fizic al circuitelor logice ale calculatorului) şi *software* (ansamblul programelor de deservire şi operare livrate odată cu calculatorul).

Calculatoarele din generația a treia apar după 1964 când firma IBM (*International Business Machines*) lansează calculatoarele din seria 360. Ele utilizează circuite miniaturizate, au memorii perfecționate, rapide, partea de software îmbogățindu-se foarte mult. Limbajele de programare se profilează pe tipuri de probleme, apare noțiunea de programare structurată. La calculatoarele din generația a treia:

- apar concepții noi în exploatare ca: *multitasking* (executarea simultană – întrețesută – a mai multor programe), regimul *time-sharing* (utilizarea aceluiaşi calculator de către mai mulți beneficiari simultan, prin acordarea de tranşe de timp succesive fiecăruia, astfel încât un beneficiar să nu blocheze în exclusivitate calculatorul);
- încep să fie folosite circuitele integrate (cu 3—10 circuite active/modul);
- apar sisteme elaborate pentru gestiunea fişierelor.

Începând cu anul 1968 se vorbeşte deja de generația a patra de calculatoare. Se consemnează perfecționări tehnologice însemnate în construcția memoriilor interne şi externe, precum şi în evoluția perifericelor. Aceste calculatoare utilizează circuite integrate cu un grad ridicat de integrare, cunoscute sub denumirea generică de *chip*-uri sau *microchip*-uri, numărul circuitelor active pe modul fiind foarte ridicat. Paşii făcuți către circuitele VLSI (*Very Large Scale Integration*) au asigurat capacitatea de prelucrare necesară construirii calculatoarelor personale, care reprezintă o parte din calculatoarele din a patra generație. Aceste aparate sunt de dimensiuni reduse, fiind eficiente şi ieftine. Deoarece nu necesită condiții ambientale speciale, ele pot fi amplasate pe un birou, în locuințe sau la diverse puncte de lucru, de exemplu în instituții, în magazine, la ieşirea din supermarket (în ghişeul de casă), în hale de producție, pe şantiere etc. Paralel cu dezvoltarea aparatelor de dimensiune redusă a crescut numărul programelor şi aplicațiilor de utilizare ce se pot rula pe asemenea calculatoare. Printre acestea se găsesc jocuri, editoare de texte, tabele de calcul, pachete pentru gestionarea bazelor de date, programe grafice, programe de comunicare etc. Aceste calculatoare nu mai sunt cumpărate, programate şi controlate doar de către specialişti şi administratori de sisteme. Ele se află deja, în sensul adevărat, la îndemâna utilizatorilor.

De la mijlocul anilor 1980 se poate vorbi şi de calculatoare din generația a cincea, un concept revoluționar introdus de fapt de către japonezi, concept ce prevedea realizarea unor echipamente de calcul prin regândirea îndrăzneață a tehnologiilor şi arhitecturilor existente. Elaborarea acestui concept a fost posibil datorită dezvoltării tehnologice coroborate cu rezultatele cercetărilor în domeniul inteligenței artificale. Până în prezent însă, rațiunile economice, dirijate şi de cerințele pieței, precum şi valoarea investițiilor existente deja în domeniul producției de componente şi de calculatoare au frânat şi au deturnat oarecum generalizarea pe această direcție de dezvoltare.

Calculatoarele timpurii nu erau prea performante, posibilitățile lor de aplicabilitate erau destul de restrânse, dar s-au întâmplat două lucruri. În primul rând calculatoarele personale au devenit din ce în ce mai eficiente, au devenit adecvate pentru rularea unor limbaje de programare cu apetit mare de memorie. Utilizarea discurilor cu suprafețe magnetice pe post de memorie a realizat accesul aparent instantaneu la cantități foarte mari de date. Pe de altă parte, calculatoarele personale pot fi conectate în rețele, realizând astfel posibilitatea conversării nemijlocite între oameni de afaceri, proiectanți etc., respectiv posibilitatea comunicării cu un calculator central care poate oferi resurse extinse.

Evoluția tehnologică a determinat deci schimbări importante în poziția şi rolul calculatoarelor în cadrul organizațiilor. La început ele erau centralizate în mare măsură şi erau folosite pentru rezolvarea unui număr restrâns de probleme (de exemplu pentru realizarea evidențelor de salarizare). Dezvoltarea tehnologiei s-a materializat în calculatoare cu dimensiuni reduse, ieftine, performante, care puteau fi amplasate pe birouri. Conform unei analize efectuate în anul 1992 s-a demonstrat că dacă automobilele *Rolls–Royce* s-ar fi dezvoltat în aceeaşi măsură ca şi calculatoarele, atunci ele ar fi consumat doar 3 litri de combustibil pentru a parcurge 1000 de km cu o viteză „normală“ de 800 km/h, prețul lor ar fi coborât sub 5 lire sterline, iar după gabaritul atins ar fi încăput într-o cutie de chibrituri. Nu se ştie însă, cine ar fi avut nevoie de asemenea automobile…

1.2 EVOLUȚIA LIMBAJELOR DE PROGRAMARE

S-a arătat că la început, la calculatoarele din prima generație, s-a utilizat *programarea numerică* în cod maşină (binar) ceea ce reprezenta o operație greoaie, necesitând cunoştințe asupra particularităților echipamentelor, antrenând şi o probabilitate considerabilă de a introduce erori în program. Pentru a scăpa de inconvenientele acestei metode au fost elaborate *limbaje simbolice simple*. Acestea conțineau de fapt o serie de mnemonice derivate din limba engleză, de genul `ADD` (adună), `MUL` (înmulțeşte) etc., alcătuind un *limbaj asamblor* legat de maşină şi necesitând traducerea în cod maşină. Prin introducerea *macro-instrucțiunilor* au apărut *limbajele simbolice evoluate*. O macro-instrucțiune corespundea la mai mult decât o operație cunoscută (executată) de calculator, fiind înlocuită în momentul traducerii cu seria de instrucțiuni maşină corespunzătoare. Timpul alocat scrierii programelor şi implicit şi riscul de a greşi s-a redus astfel considerabil. Programul traducător al acestor limbaje se numea autocodificator. Limbajele asamblor cu autocodifica-tor au coexistat în perioada 1958—1964, dar s-au folosit şi ulterior la programarea

calculatoarelor din generația a treia cu menținerea denumirii unificate de *limbaj asamblor* (sau *limbaj macroasamblor*).

Un salt calitativ în domeniul limbajelor de programare îl constituia apariția *limbajelor algoritmice* (sau *limbaje procedurale*) nelegate de calculator. De fapt independența nu era totală, fiind necesare mici corecturi, adaptări, funcție de particularitățile calculatoarelor utilizate, aceste modificări nefiind însă esențiale ca volum. Ca orice limbaj, şi cele algoritmice (procedurale) se caracterizează printr-un vocabular şi prin reguli de sintaxă.

Vocabularul este alcătuit dintr-un ansamblu de cuvinte cheie (preluate şi adaptate de regulă din limba engleză), iar numele variabilelor sunt date de programator (respectând anumite reguli). Limbajele algoritmice permit scrierea algoritmilor după care urmează să fie soluționată problema abordată, sub formă de *instrucțiuni* (fraze cu un conținut semantic bine precizat). Prin *semantica* limbajului se înțelege ansamblul regulilor de corespondență sau de interpretare corespunzătoare cuvintelor cheie, respectiv grupelor (blocurilor) de cuvinte cheie. Frazele limbajului (rândurile de instrucțiune) vor fi alcătuite deci din combinații de cuvinte cheie şi nume de variabile, după anumite reguli. *Sintaxa* limbajului stabileşte combinațiile posibile de cuvinte cheie, nume de variabile precum şi folosirea punctuației.

Primul limbaj algoritmic de nivel înalt este considerat FORTRAN (denumirea provine de la *FORmula TRANslation system*) apărut în 1954, însă merită să amintim şi alte limbaje consacrate de acest gen, cum ar fi COBOL (*Common Business Oriented Language* – apărut în 1959 ca urmare a dezvoltării limbajelor B-O din 1957 şi Flow-Matic din 1958, prima standardizare fiind cea din 1961, derivată din CODASYL – *Conference on Data Systems Language*, 1959); ALGOL (*ALGorithmic Language* – 1958, apărut pe baza limbajului Fortran prin combinarea limbajelor JOVIAL, NELIAC, BALGOL şi MAD) fiind limbajul din care au fost dezvoltate ulterior limbajele CPL (predecesorul din 1963 al limbajului C care a apărut în 1971), PL/I (1964), Simula (1964) Pascal (1970) etc.; sau BASIC (*Beginners All-purpose Symbolic Instruction Code* – 1964).

Paralel cu dezvoltarea limbajelor procedurale au apărut şi primele limbaje funcționale de programare. Structura acestora este oarecum detaşată de noțiunea convențională de algoritm (în sensul utilizat la limbajele algoritmice), ea fiind însă mai apropiată modului uman de gândire. Ca şi reprezentare consacrată pentru această categorie trebuie să amintim limbajele LISP (*LISt Processing* – apărut în 1958) şi PROLOG (*PROgramming in LOGic* – apărut în 1970), existând foarte multe variante ale acestora, pe lângă alte limbaje funcționale noi. Deşi aceste limbaje sunt asociate în general cu cercetările din domeniul inteligenței artificiale, ele au aplicabilitate şi în domeniul ingineriei.

Având în vedere că în rezolvarea unor categorii de probleme specifice anumitor domenii de activitate se întâmpinau dificultăți chiar utilizând limbaje algoritmice avansate, orientate pe sisteme, dificultățile apărând fie la introducerea datelor fie din folosirea unor cuvinte cheie diferite de limbajul tehnic consacrat în domeniul respectiv, au fost dezvoltate limbaje (şi compilatoare) *orientate pe problemă*. Astfel, în domeniul mecanicii construcțiilor pentru

determinarea eforturilor în structurile alcătuite din bare s-a elaborat în S.U.A. limbajul STRESS (*STRuctural Engineering System Solver* – 1963), care s-a bucurat de o răspândire largă, iar în 1966 a fost elaborat ICES (*Integrated Civil Enginnering Systems*), un sistem integrat de rezolvare a problemelor de construcții care conținea următoarele limbaje orientate pe probleme:

- COGO (*Coordinate GeOmetry*) pentru rezolvarea problemelor geometrice,
- STRUDL (*STRUctural Design Language*) pentru analiza şi proiectarea structurilor,
- PROJECT (*PROJect Evaluation and Coordination Techniques*) pentru problemele schițării, rețele, drum critic etc.,
- SEPOL (*SEttlement Problem Oriented Language*) pentru calcule de tasări în mecanica solului,
- ROADS (*Roadway Analysis and Design System*) pentru amplasarea, alinierea drumurilor, râurilor etc.,
- BRIDGE pentru analiza şi proiectarea podurilor,
- TRANSET (*TRANSportation Evaluation Techniques*) pentru analiza rețelelor, fluxurilor de transport.

Limbaje similare orientate pe probleme au mai fost elaborate şi în alte țări. Preocupări în această direcție şi realizări remarcabile s-au obținut şi la noi în țară. Amintind doar câteva exemple specifice din domeniul analizei structurale: în cadrul C.O.C.C. Bucureşti a fost elaborat limbajul SISBAR (autor L. Dogaru) pentru analiza generală a structurilor alcătuite din bare, iar în cadrul Universității Tehnice din Cluj-Napoca a fost dezvoltat limbajul SICAM (*Sistem Interpretativ de Calcul Automat Matriceal*, autor F. Gobesz) pentru analiza structurilor în formularea matriceală.

Un salt calitativ considerabil în domeniul dezvoltării limbajelor procedurale a reprezentat apariția noțiunii de *orientare pe obiecte*. Astfel, pe lângă apariția unor limbaje noi, au fost dezvoltate variante corespunzătoare şi în cadrul limbajelor existente şi consacrate (Fortran 90, OO Cobol, C++, Borland Pascal etc.). Prin tendințele de detaşare față de varietatea sistemelor de operare au apărut *limbajele interpretate* şi *script*-urile (acestea nenecesitând compilatoare în sensul cunoscut, fiind convertite în cod maşină doar în momentul executării programului cu ajutorul unui sistem de interpretare specific sistemului de operare). Dintre aceste limbaje merită să amintim Java (apărut în 1995 din limbajul Oak care s-a născut în 1991 din limbajele Cedar, Objective-C, C++ şi SmallTalk-80) respectiv JavaScript (apărut în 1995 din LiveScript care la rândul lui a derivat din Cmm, acesta din urmă dezvoltându-se din limbajul ISO C – cunoscut şi ca C90 – respectiv din C++).

În cele ce urmează vom sintetiza istoria limbajului Fortran, având în vedere orientarea acestei lucrări. Limbajul a apărut în noiembrie 1954, dezvoltarea lui fiind inițiată de către o echipă remarcabilă de programatori de la IBM (sub conducerea lui John Backus). Din anul 1957 se poate vorbi de limbajul şi compilatorul Fortran I, destinate inițial calculatorului IBM 704. Fiind primul limbaj algoritmic apărut, oferind o serie de avantaje față de limbajul de asamblare, a fost adoptat foarte repede de comunitățile ştiințifice din toate domeniile.

Acest succes a concurat la dezvoltarea firească a limbajului, apărând variante ca Fortran II în 1957, Fortran III în 1958, Fortran IV în 1962 precum şi convertoare pentru a menține compatibilitatea cu programele scrise în versiunile anterioare. În luna mai a anului 1962 a devenit primul limbaj de nivel înalt standardizat oficial, iar în 1966 a fost adoptat primul standard ANSI (*American National Standards Institute*) referitor la Fortran IV (acesta primind denumirea oficială de Fortran 66 ANS). Această versiune a fost urmată de Fortran V, respectiv de alte tentative de dezvoltare.

Până la mijlocul anilor 1970 aproape toate calculatoarele mici şi mari dispuneau de compilatoare pentru Fortran 66, acesta fiind limbajul de programare cel mai răspândit. Cu trecerea timpului şi cu dezvoltarea limbajului, definiția standardizată a fost actualizată în 1978 (ANSI X3.9), ea fiind adoptată şi de ISO (*International Standardization Organization*) în 1980 ca normă internațională, apărând astfel versiunea cea mai cunoscută a limbajului, sub denumirea Fortran 77. Caracterul relativ conservator al acestui standard a lăsat limbajului un număr de facilități ce păreau din ce în ce mai depăşite de către noile limbaje de programare apărute (unele derivate în parte chiar din Fortran), aşa că în mod firesc din anii 1980 s-a trecut la dezvoltarea acestuia (sub numele Fortran 8x). Deşi limbaje ca Algol, Basic, Pascal şi Ada erau mai uşor de învățat şi de utilizat, ele nu se puteau compara cu eficiența dovedită de Fortran în ceea ce privea viteza de calcul şi compactitatea codului generat.

A treia standardizare a limbajului Fortran s-a produs abia în 1991, versiunea limbajului primind numele Fortran 90. Această întârziere a dus la accentuarea competiției în privința popularității limbajelor de programare, având efecte negative asupra limbajului Fortran. Chiar adepți convinşi ai acestui limbaj au migrat către limbaje ca C sau C++, în ciuda lipsei de performanțe în calcule ale acestora. Prin includerea completă a versiunii anterioare (Fortran 77) alături de noutățile introduse, oferind flexibilitate şi performanțe remarcabile, noua versiune a limbajului Fortran a reuşit să se impună în rândul programatorilor, permițând realizarea programelor în moduri adecvate mediilor moderne. Prin extinderea limbajului în direcția procesării paralele s-a standardizat HPF (*High Performance Fortran*) în 1993. În anul 1996 s-a dat publicității a patra standardizare a limbajului Fortran, acesta primind numele Fortran 95, la care s-au adăugat încă două rapoarte tehnice elaborate sub egida ISO. Ultima versiune a limbajului este cea din anul 2000, purtând numele sugestiv Fortran 2000.

1.3 NOȚIUNI REFERITOARE LA REPREZENTAREA DATELOR

Aşa cum s-a menționat în primul capitol, *date*le reprezintă modele manevrabile ale informațiilor. Calculatoarele personale stochează aceste date pe medii magnetice şi optice, utilizând structuri logice arborescente. Pentru a ne face o idee despre aceste structuri, trebuie să lămurim câteva noțiuni fundamentale.

Din motive tehnologice calculatoarele folosesc un set limitat de valori pentru reprezentarea datelor. Unitatea elementară de memorie utilizată pentru date se numeşte *bit*. Aşa cum un

magnet poate dispune de doi poli, un circuit electric poate avea curent sau nu, starea unui *bit* poate fi notată cu două valori distincte: 0 sau 1. Deoarece cele două valori oferă o plajă foarte limitată, aceste entități elementare sunt grupate în *octeți* (*bytes*), doi octeți formând o entitate adresabilă ce poartă denumirea de *cuvânt*. Manevrarea teoretică a valorilor binare fiind greoaie şi consumatoare, se utilizează reprezentarea în cifre *hexadecimale* a acestora, derivată din expresiile greceşti *hexa* (6) şi *deca* (10) care compuse, marchează numărul şaisprezece (16).

Tabel comparativ cu reprezentarea unor valori în sisteme de numărare diferite:

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Binar (pe 4 biți)	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Privind tabelul de mai sus se poate observa uşor faptul că pe un octet (pe opt biți) pot fi reprezentate în sistemul binar două cifre din sistemul hexadecimal, deci gruparea biților în octeți nu este întâmplătoare. Astfel putem înțelege de ce se consideră mai comodă utilizarea cifrelor hexadecimale în locul celor decimale pe calculatoare. Cifrele decimale fiind de la 0 la 9, s-ar consuma câte 4 biți pentru reprezentarea uneia, combinațiile binare din plaja 1010—1111 fiind inaccesibile. Deci, dacă am încerca să reprezentăm într-un octet (*byte*) valorile prin cifre decimale, nu am putea scrie o valoare mai mare de 99, pierzând astfel plaja valorilor dintre 100 şi 255 (ceva mai mult de jumătatea numerelor reprezentabile). Prin utilizarea sistemului hexadecimal putem reprezenta două cifre pe un octet, deci patru cifre pe un cuvânt (FFFF scris în hexa însemnând valoarea 65536 în sistemul decimal). Dacă se consideră şi noțiunea de semn, atunci, în maniera anterior discutată, pe doi octeți pot fi reprezentate valorile corespunzătoare unei notații în sistemul decimal de la –65535 până la +65535 (în acest caz FFFF scris în hexa va însemna valoarea –1 în sistemul decimal obişnuit).

Pentru a putea manevra eficient datele, ele trebuie să fie grupate şi adresabile. În acest sens vom defini două noțiuni des întâlnite în utilizarea calculatoarelor:

1. *Fişierul* este o colecție organizată de date ce se tratează uniform. (După părerea autorilor, aceasta este cea mai scurtă şi cea mai pertinentă definiție pentru asemenea colecții, cuprinzând patru caracteristici esențiale: datele sunt conținute în fişiere, sunt organizate, se pot efectua operații asupra lor, toate datele dintr-un fişier pot suporta aceleaşi operații.)

2. *Directorul* (se utilizează şi denumirile: *folder*, *dosar*) este o colecție de referințe organizate ce se tratează uniform. Definiția este asemănătoare cu cea a fişierelor, totuşi există diferențe semnificative pe care le vom exemplifica în cele ce urmează.

Dacă ne imaginăm, pentru exemplificarea definițiilor anterioare, o cămară (pe post de memorie de stocare) în care depozităm alimente (privite ca şi date) şi ambalaje, atunci putem face următoarele analogii:

Ținând, de exemplu, mere într-o cutie, putem privi acea cutie ca pe un fişier de mere, iar o altă cutie în care ținem sticle goale, ca un fişier de sticle goale. Evident, atât merele cât şi

sticlele vor alcătui colecții organizate (dupa forma cutiilor, după legea gravitației etc.) care pot suporta aceeaşi prelucrări. Chiar daca momentan nu avem mere în cutia respectivă, cutia va fi tot un fişier de mere atâta timp cât nu i se schimbă caracteristicile, în acest caz putând vorbi despre un fişier gol de mere. Ceea ce este important de observat: cutiile pot conține la nivel fizic entitățile pentru care sunt destinate.

Dacă dorim să avem o evidență referitoare la aceste cutii, vom nota într-un carnețel numele şi poziția lor. Un asemenea carnețel va conține astfel doar referințe la cutii (la fişiere) reprezentând un director (folder sau dosar), fără a conține fizic cutiile (fişierele) respective. Bineînțeles, un asemenea carnețel poate conține şi alte însemnări, de exemplu referitoare la alte carnețele (în care sunt notate denumirile şi pozițiile altor cutii). Asemenea referințe pot fi percepute ca şi subdirectoare. Prin acest mod de evidență se poate crea o structură arborescentă similară cu cea existentă în mediile de stocare ale datelor.

Interfețele utilizator ale sistemelor de operare pot induce neclarități referitoare la aceste concepte, sugerând prezența fizică a fişierelor în directoare pentru a simplifica operațiile. De cele mai multe ori, ştergând o înregistrare se va obține de fapt marcarea specială a acesteia (referința devine ascunsă iar zona alocată se poate suprascrie) şi nu ştergerea fizică.

Fişierele pot fi de tipuri diferite, în funcție de conținutul lor. Ele sunt împărțite de regulă în două categorii mari: fişiere cu caractere afişabile şi fişiere cu coduri neimprimabile (dacă merele se pot mânca, încercarea de a consuma sticle goale poate avea urmări imprevizibile). Fişierele cu coduri neimprimabile pot fi executabile sau nu. În cazul în care ele sunt executabile, poartă numele generic de *program* sau de *comandă externă* (*comenzile interne* sunt cele conținute de programele încărcate în memoria internă a calculatorului). Un caz particular este reprezentat de fişierele cu caractere afişabile care sunt „executabile“ (de regulă acestea sunt interpretate). Acestea conțin de fapt linii de comenzi într-o succesiune secvențială şi poartă denumirea generică de *fişiere de comandă*.

1.4 NOȚIUNI REFERITOARE LA PROGRAME

În capitolul anterior am dat o definție pentru noțiunea de program, din punctul de vedere al fişierelor. Programul poate fi definit şi ca mulțimea comenzilor (instrucțiunilor) selecționate de către programator din mulțimea de instrucțiuni puse la dispoziție prin platforma de calcul. În cele ce urmează vom discuta şi alte aspecte legate de această noțiune.

Se ştie foarte bine că dacă pentru soluționarea unor probleme dintr-un domeniu limitat dispunem de o descriere clară şi bine definită a procedurii de rezolvare, atunci această muncă (în unele cazuri cu pronunțat caracter repetitiv) poate fi încredințată unui sistem automat de calcul. Cele mai multe programe sunt scrise sub forma unor algoritmi, acestea conținând practic lista instrucțiunilor în ordinea corespunzătoare executării lor de către calculator. De cele mai multe ori un asemenea algoritm utilizează şi date care sunt organizate într-o anumită formă, în fişiere separate, externe. Programul de calculator poate fi privit astfel şi ca un pachet alcătuit din *date + algoritm*.

De exemplu, un program pentru calculul salariilor poate fi descris în felul următor: caută prima persoană din înregistrări, citeşte salariul brut al persoanei găsite, determină treapta de impozitare în care se încadrează, calculează impozitul, caută toate celelalte scăzăminte (contribuții la fonduri de asigurări, cotizații etc.) şi determină salariul net. Toate aceste elemente ajung sub o formă prestabilită pe fluturaşul de salarizare şi pe statul de plată. După prelucrarea datelor primului angajat, sistemul pentru calcularea salariilor continuă determinarea salariilor pentru ceilalți angajați, indiferent dacă numărul lor este de ordinul zecilor sau al sutelor. Algoritmul nu se schimbă la nici un angajat. Sistemele de acest gen se bazează pe tehnica de calcul timpurie proiectată pentru rezolvarea problemelor numerice repetitive, complet plictisitoare. Aplicațiile mai recente oglindesc deja şi schimbările tehnologiei. Răspândirea utilizării calculatoarelor personale a condus la apariția unor aplicații mai mici de salarizare, bazate pe tabele de calcul, care sunt compuse din şiruri şi coloane de numere manipulate cu ajutorul sistemului de calcul. Pachetele de programe care au fost vândute în numărul cel mai mare sunt tabelele de calcul şi procesoarele de texte. Oamenii însă nu folosesc numai cuvinte şi cifre, ci manevrează şi o cantitate din ce în ce mai mare de informații referitoare la acestea. Acest fapt a condus la apariția şi dezvoltarea bazelor de date. Bazele de date înmagazinează o cantitate enormă de informații sub formă structurată, utilizatorul având acces la informația dorită oricând, în orice formă dorită. Un exemplu foarte bun pentru ilustrarea celor amintite poate fi o bază cu date despre angajații din cadrul unei firme. În structura bazei de date fiecărui angajat îi corespunde un articol, acesta conținând informațiile utile referitoare la angajat, de exemplu: numele, adresa, data naşterii, salariul de încadrare, locul de muncă etc. Utilizatorul poate întocmi, din informațiile conținute de către baza de date (de regulă prin folosirea unui limbaj de interogare), o situație pentru analiză despre angajați, ori de câte ori este nevoie. Poate obține rapoarte de genul: lista angajaților cu salariul peste *X* lei, cu vechime peste *Y* ani, sau domiciliați într-o anumită zonă. Ceea ce putem obține din baza de date este limitat doar de către informația înmagazinată şi de către limbajul de interogare.

Din cele prezentate mai sus se desprinde ideea că un pachet de programe poate avea o alcătuire complexă. În această carte vom aborda doar realizarea unor aplicații de tip consolă cu ajutorul limbajului *Fortran* şi al compilatorului *g77*, fără a discuta despre conceperea şi generarea interfețelor grafice sau despre detaliile şi specificitățile caacteristice diverselor sisteme de operare. Etapele de programare sunt în general sintetizate prin 3 faze: concepția (nivelul logic de rezolvare a problemei cu elaborarea sau alegerea algoritmului corespunzător), codificarea (transcrierea algoritmului într-un limbaj de programare, accesibil calculatorului), testarea şi implementarea (verificarea corectitudinii cu date de test şi punerea la punct a programului). Aceste faze pot fi realizate atât într-o formă empirică cât şi în manieră structurată, mai eficientă decât prima variantă (permițând dezvoltarea şi testarea modulată). Preferând abordarea structurată, în locul celor 3 faze generice amintite prezentăm următoarea succesiune de etape, considerate necesare realizării unui program:

- recunoaşterea şi definirea problemei de abordat;
- alegerea şi descrierea metodei (algoritmului) de rezolvare;
- scrierea sursei programului într-un limbaj de programare;
- compilarea (traducerea în cod maşină a) sursei;

- realizarea legăturilor cu modulele de bibliotecă ale mediului de programare;
- rularea şi testarea programului creat.

De ce sunt necesare asemenea etape de parcurs?

În primul rând, fără recunoaşterea problemei nu avem ce rezolva. Prin definirea problemei se pot determina datele de intrare necesare şi suficiente, precum şi rezultatele urmărite a fi obținute.

Aproape orice problemă poate fi abordată şi rezolvată pe mai multe căi. Există o varietate foarte mare de metode studiate şi validate în diferite domenii, din această cauză doar rareori se impune crearea sau inventarea unor metode complet noi. Alegerea unei metode trebuie să țină seama atât de caracteristicile problemei, cât şi de echipamentele şi instrumentele informatice accesibile programatorului, precum şi de generalitatea şi aplicabilitatea soluțiilor urmărite. Descrierea pertinentă şi clară a metodei permite şi eliminarea unor erori pe lângă modularizarea procedeelor în vederea eficientizării programării. În general metodele pot fi clasificate în două categorii: *metode directe* şi *metode indirecte*. Metodele directe sunt cele care ne conduc la rezultat într-un număr finit de paşi. Metode indirecte sunt considerate cele care ar necesita parcurgerea unui număr infinit (mai corect nedefinit) de paşi pentru a obține un rezultat exact. Deşi un număr infinit de paşi nu pare o soluție practică, sunt cazuri în care nu avem la îndemână decât asemenea metode. Ele se aplică prin alegerea unor toleranțe față de rezultatul urmărit (prin aprecierea convergenței, limitarea unor abateri etc.) ceea ce conduce la un număr necunoscut de paşi, dar finit. Condiționarea metodelor este un alt concept de care se ține seama la alegere. Sunt considerate *bine condiționate* acele metode la care perturbările mici în datele inițiale nu conduc la alterarea deranjantă a rezultatelor. Metodele în care perturbările inițiale au ca efect deranjarea rezultatelor, se consideră *slab condiționate* şi au de regulă o aplicabilitate mai redusă, necesitând un control mai strict al validărilor. Experiența şi rutina sunt factori importanți în această etapă.

Limbajul de programare şi mediul în care se scrie sursa programului se recomandă a fi alese după specificul problemei abordate şi a metodei de soluționare pentru care s-a optat. Sursa poate fi compusă din mai multe *unități sursă de program* (uneori denumite module) care pot fi redactate chiar utilizând fişiere separate. Ea trebuie să fie conformă rigorilor impuse de limbajul de programare ales şi de mediul de dezvoltare folosit (sistemul de operare, mediul de programare etc.). În limbajul Fortran se poate creea doar un singur program sursă principal care poate fi combinat însă cu diverse *subunități de program* (blocuri de date, subprograme, funcții externe etc.) scrise chiar în fişiere separate.

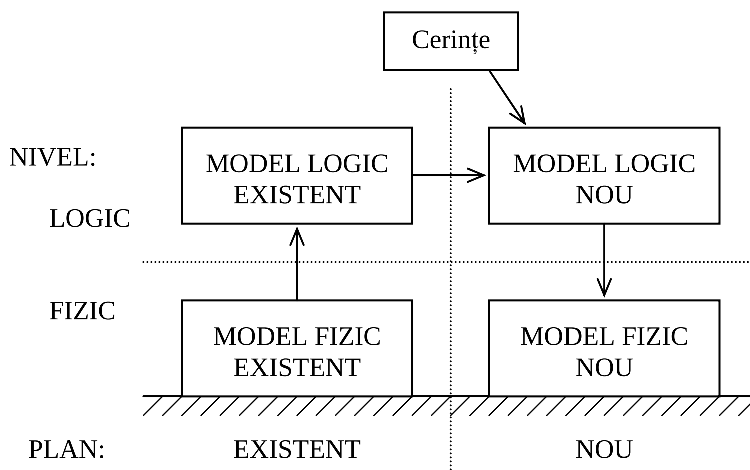
Deoarece textul rezultat în urma scrierii sursei se materializează prin crearea unor fişiere cu caractere afişabile (imprimabile) ce nu sunt interpretabile de către calculator, instrucțiunile conținute vor trebui traduse în coduri înțelese de maşină. Acest lucru este realizat prin compilarea sursei, etapă în care se poate face şi verificarea sintaxei (verificarea semanticii însă nu poate fi efectuată de *compilator*). O asemenea traducere însă nu realizează şi transformarea sursei în program (*fişier executabil*), creând doar *imagini obiect* ale surselor

(fişiere neexecutabile dar care conțin coduri maşină). Fişierele cu date de intrare nu trebuie compilate pentru că ele nu conțin instrucțiuni ci doar valori ce vor fi utilizate ca atare, independente de limbajul de programare utilizat.

Pentru a obține fişiere executabile (*programe* propriu-zise), trebuie prelucrate legăturile generate în cadrul imaginilor obiect, ținând cont şi de bibliotecile necesare. Acest lucru se realizează de regulă prin procedee automate ale mediului de dezvoltare ales, în urma opțiunilor exprimate de programator. Unele medii de dezvoltare oferă această facilitate în mod transparent, aparent împreună cu opțiunea de compilare. În această etapă este posibilă şi definirea unei structuri preferențiale explicite a programului prin *segmentare* (stabilirea succesiunii de încărcare în memorie şi a dependințelor modulelor create).

Rularea şi testarea programului realizat, menționate ca etapă finală, sunt foarte importante pentru verificarea performanțelor obținute şi a corectitudinii rezultatelor în diverse ipostaze. Toate etapele descrise pot şi se recomandă a fi supuse unor rafinări sau unor abordări repetitive în scopul depistării şi eliminării erorilor posibile.

Ca orice proces de proiectare, conceperea şi dezvoltarea aplicațiilor informatice se supune unor reguli şi se poate privi ca un ciclu. Din acest punct de vedere, orice asemenea activitate presupune parcurgerea nu numai a unor etape fizice, ci şi a unora logice, pe niveluri şi în planuri diferite:



Pentru a putea altera situația existentă, avem nevoie de un model corespunzător. Prin formularea acestuia putem determina schimbările necesare. Prima etapă va însemna delimitarea şi măsurarea caracteristicilor într-un plan fizic, din care prin abstractizare se va naşte un model logic. Intervenind asupra acestuia prin concepția schimbărilor propuse, considerând şi cerințele externe, se va obține un nou model logic, care la rândul său se va

transforma într-un model fizic nou. Deci, pornind de la o situație existentă, am ajuns la o situație nouă. Caracterul ciclic al acestui proces este evident în acest moment, deoarece situația nou creată va deveni una existentă (din momentul implementării), care la rândul ei va fi la un moment dat alterată printr-un proces similar.

Era o vreme când programatorii erau considerați artişti, şi asta în principal datorită manierelor personale de lucru. Trecerea şi în acest domeniu de la artizanat la industrializare a constituit o constrângere, derivând arta scrierii aplicațiilor în ştiința programării. Această transformare urmărea şi trecerea de la microeficiență la macroeficiență în domeniu, ceea ce a necesitat un compromis între resursele necesare (spațiu de memorie, timp de rulare, costuri de realizare şi de întreținere) şi fiabilitate. Toate acestea au condus în mod firesc către modularitatea programelor şi către principiile structurării, din 1968 fiind recunoscută şi disciplina ingineriei de soft.

Bazele programării structurate au fost enunțate în 1969 de către Dijkstra (prin teorema de structură), fiind aplicate pentru prima dată în cadrul corporației IBM la un proiect pentru determinarea orbitelor definitive ale sateliților tereştri, cu un rezultat deosebit (în decursul a 6 luni s-a reuşit de către Mills – un "superprogramator" – scrierea şi validarea în limbajul PL/1 a 50 de mii de instrucțiuni, ceea ce până atunci reprezenta norma obişnuită pentru 30 de oameni de-a lungul unui an).

CAPITOLUL 2: ALGORITMI

2.1 NOȚIUNI DESPRE ALGORITMI

Un algoritm poate fi definit ca mulțimea finită de reguli care indică o succesiune de operații pentru rezolvarea în mod mecanic (sau automat) a unui anumit tip de probleme. Principiile de bază, valabile atât la analiza/definirea unei probleme, cât şi la celelalte etape de realizare (prezentate în capitolul precedent) pot fi sintetizate în următorul mod:

- Concepția la orice nivel prin descompunere – toate metodele utilizate în mod curent respectă acest principiu, de exemplu: principiul KISS (*Keep It Stupid Simple*) din metoda Yourdon, principiul proiectării ierarhice din metoda Constantine, principiul detaliilor din metoda Warnier, principiul primitivelor din metoda Dijkstra etc.;
- Realizarea analizei/concepției prin descompunere (de sus în jos) combinată cu recompunerea (de jos în sus) în faza de realizare;
- Structura datelor determină structura prelucrărilor (aplicarea metodelor bazate pe structuri de date);
- Aplicarea unui proces iterativ de verificare (revenire, chiar cu rafinare dacă se impune) în toate fazele de elaborare a unui produs informatic.

În general se recomandă utilizarea metodelor de programare modulară şi structurată. Aceaste metode reprezintă o manieră de a concepe şi codifica programe astfel încât să fie uşor de înțeles şi de modificat. Este nevoie de o anumită experiență, dar aceasta se acumulează relativ uşor, metoda rezultând din procesul de organizare al gândirii care duce la o expresie inteligibilă a procesului de calcul într-un timp rezonabil, fiind considerată şi „arta simplității“ sau „reîntoarcerea la bun-simț“. Cele mai importante proprietăți ale unui algoritm sunt considerate următoarele:

- Definibilitatea – fiecare pas trebuie să fie foarte bine precizat, atât ca şi conținut cât şi ca poziție.
- Realizabilitatea – obținerea rezultatului în timp util, cu resurse corespunzătoare.
- Finitatea – aplicarea algoritmului să ne conducă la rezultat după un număr finit de paşi.
- Generalitatea – algoritmul să se aplice unei întregi familii (clase) de probleme şi nu doar unui caz izolat (utilizarea variabilelor).
- Automatismul – să necesite cât mai puține intervenții umane după lansarea în execuție.

Elementele caracteristice ale unui algoritm sunt:

- datele (informația vehiculată) – date de intrare, de ieşire, intermediare;
- operațiile – operații de atribuire, de calcul, de decidere, de salt, de citire sau de scriere, de deschidere sau de închidere fişiere etc.;
- paşii – descriu regulile algoritmului.

2.2 TEHNICI ŞI INSTRUMENTE DE REPREZENTARE

Specificarea unui proces înseamnă elaborarea unei descrieri concise şi complete a transformării efectuate de acesta. Primul principiu enunțat în subcapitolul anterior: concepția la orice nivel prin descompunere, a condus la elaborarea descrierilor pentru procesele elementare (numite şi *primitive funcționale*). Aceste descrieri poartă denumirea de minispecificații, reprezentând regulile de transformare a elementelor datelor de intrare în elemente ale datelor de ieşire la nivelul procesului elementar. Descrierea algoritmilor trebuie să respecte aceeaşi principii pe care le întâlnim la minispecificații. Ea trebuie să conțină aspectele logice (ceea ce se realizează în cadrul procesului) şi nu aspectele fizice (cum se realizează ceea ce se procesează) într-o manieră clară, concisă şi completă, fără a eluda caracteristici esențiale ale procesului. Metodele de descriere a proceselor pot fi prin:

- text obişnuit: descriere imprecisă, redundantă, abundă în elemente nesemnificative, greu de scris, foarte greu de înțeles şi aproape imposibil de actualizat;
- limbaj natural structurat: descriere precisă, neredundantă, greu de scris dar uşor de înțeles şi se poate actualiza;
- pseudocod: descriere foarte precisă, vocabular redus şi uşor de controlat (dar greu de înțeles pentru nespecialişti);
- tabele de decizie: descriere foarte precisă, uşor de transformat în programe, greu de scris (şi relativ greu de înțeles de către nespecialişti);
- instrumente grafice: (arbori, grafuri, scheme etc.) cu caracteristici şi facilităţi diferite, considerate însă ca fiind cele mai eficiente din punct de vedere al comunicării concise.

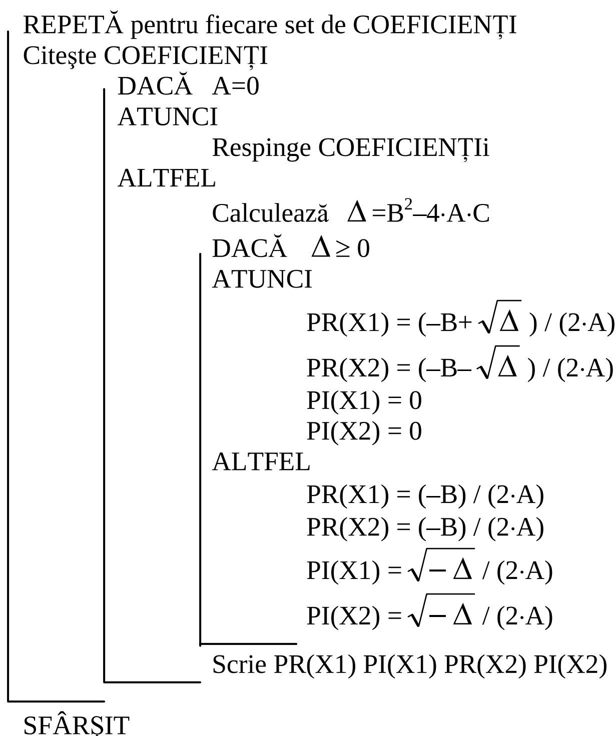
În cele ce urmează vom trece în revistă câteva metode şi instrumente de reprezentare. Textul obişnuit (limbajul natural scris sau înregistrat) nu prezintă un interes deosebit în cazul de față (deşi redactarea acestei cărți s-a făcut în mare parte prin această metodă), aşa că vom sări peste el şi vom începe direct cu varianta structurată a acestuia.

2.2.1 Limbajul natural structurat

Deficiențele limbajului natural în ceea ce priveşte descrierea proceselor (algoritmilor) au condus la construirea unui limbaj cu o sintaxă riguroasă dar cu un vocabular mai sărac, prin renunțarea la calificatori considerați nesemnificativi (adjective şi adverbe), la alte moduri ale verbelor decât cele imperative, la punctuația excesivă, la adnotări şi prin impunerea folosirii unor propoziții enunțiative, simple. Ca şi definiție, putem spune că limbajul natural structurat (prescurtat: LNS) este un limbaj simplificat care îmbină vocabularul limitat din limbajul gazdă cu sintaxa limbajelor de programare structurată.

Vocabularul limbajului natural structurat este constituit din verbe cu înțeles neambiguu (la modul imperativ), termeni ce descriu obiecte şi atribute precise, cuvinte “rezervate” pentru formulare logică (*dacă*, *atunci*, *altfel*, *repetă* etc.). Sintaxa enunțurilor este limitată la

propoziții enunțiative simple, construcții decizionale (cu două posibilități: *da* sau *nu*), construcții de repetiții, precum şi combinații ale acestor trei. Pentru claritatea exprimării şi folosirea unui set redus de simboluri, terminarea unei construcții decizionale sau de repetiție se marchează cu o linie verticală care începe la începutul construcției şi se încheie după ultimul enunț al acesteia. Pentru exemplificare, iată descrierea rezolvării ecuației de gradul doi ($A \cdot X^2 + B \cdot X + C = 0$) în LNS, calculând partea reală (PR) şi imaginară (PI) a rădăcinilor:



```
REPETĂ pentru fiecare set de COEFICIENȚI
Citeşte COEFICIENȚI
        DACĂ   A=0
        ATUNCI
                Respinge COEFICIENȚIi
        ALTFEL
                Calculează  Δ=B²–4·A·C
                DACĂ   Δ ≥ 0
                ATUNCI
                        PR(X1) = (–B+ √Δ ) / (2·A)
                        PR(X2) = (–B– √Δ ) / (2·A)
                        PI(X1) = 0
                        PI(X2) = 0
                ALTFEL
                        PR(X1) = (–B) / (2·A)
                        PR(X2) = (–B) / (2·A)
                        PI(X1) = √–Δ / (2·A)
                        PI(X2) = √–Δ / (2·A)
                Scrie PR(X1) PI(X1) PR(X2) PI(X2)
SFÂRŞIT
```

Printre avantajele limbajului natural structurat putem menționa faptul că poate fi folosit în orice etapă din ciclul de viață al proiectului (analiză, proiectare logică, proiectare tehnică) fiind concis şi uşor de înțeles, cu o sintaxă simplă, constituind un bun limbaj "intermediar" fiind apropiat atât de limbajul natural cât şi de limbajele evoluate de programare, şi poate fi redactat şi întreținut cu editoare de text sau chiar editoare de programare.

Ca şi dezavantaje trebuie să menționăm că realizarea unui set de reguli sintactice şi a unui vocabular adecvat pentru un LNS este o activitate de durată cu implicarea unei responsabilități majore din partea autorilor; părând mai formalizat decât este în realitate este acceptat mai greu de către analişti şi programatori; refacerea iterativă a descrierilor (în etapa de analiză mai ales) este consumatoare de timp şi cere efort; şi nu în ultimul rând, o bună descriere în LNS a procesului nu garantează corectitudinea (un fenomen prost înțeles poate fi exprimat la fel de uşor în şi de coerent în LNS ca şi în limbaj natural).

2.2.2 Pseudocodul

Este foarte asemănător cu LNS, fiind tot o reprezentare a paşilor algoritmului sub formă de propoziţii simple şi construcţii decizionale şi repetitive. Diferenţa majoră faţă de LNS constă în vocabularul creat din limba engleză şi din apropierea mai accentuată prin sintaxă de limbajele de programare structurate de nivel înalt. În următorul tabel se poate observa diferenţa dintre limbajul natural structurat, pseudocod şi schema logică (prin prisma a trei primitive funcţionale: decizia logică, repetiţia postcondiţionată şi repetiţia precondiţionată).

Tabel cu 3 exemple de structuri logice elementare

Descriere în LNS	Descriere în pseudocod	Reprezentare în schemă logică
DACĂ C ATUNCI Procedura A ALTFEL Procedura B	IF C THEN DO A ELSE DO B ENDIF	Da C Nu A B
REPETĂ până când C Procedura A	REPEAT UNTIL C DO A ENDREPEAT	A C Nu Da
REPETĂ cât timp C Procedura A	WHILE C DO A ENDWHILE	Da C A Nu

2.2.3 Tabele de decizie

Este bazată pe identificarea şi codificarea condiţiilor logice şi a acţiunilor ce intervin în urma unor combinaţii ale acestor condiţii. Acest instrument se poate folosi în orice etapă din dezvoltarea unui program, cu toate că iniţial era folosit doar pentru descrierea procedurilor. Există şi generatoare de programe ce acceptă ca şi surse de intrare tabele de decizie. Tabelele pot fi împărţite în patru zone cu semnificaţii distincte: definirea condiţiilor, definirea acţiunilor, combinaţia condiţiilor, respectiv efectul acestor combinaţii în acţiuni. În funcţie de tipul conţinutului zonelor a treia şi a patra, putem vorbi de tabele cu

intrări/ieşiri simple sau multiple. În cazul în care sunt marcate în cadrul acțiunilor salturi către alte tabele de decizie, putem vorbi de tabele imbricate (înlănțuite).

În general, la alcătuirea unei tabele de decizie trebuie respectate următoarele principii:

- condițiile sunt prioritare față de acțiuni, acestea din urmă find selectate în funcție de combinațiile condițiilor;
- condițiile trebuie să fie independente (unele față de altele);
- orice combinație de condiții va conduce la un set de acțiuni definit.

Pornind de la aceste principii, alcătuirea unei tabele de decizii se realizează în următoarea secvență:

- se identifică şi se definesc toate condițiile (în prima zonă);
- se definesc toate acțiunile (ce vor alcătui a doua zonă);
- se completează toate combinațiile valide ale condițiilor (în a treia zonă);
- se alcătuiesc regulile prin marcarea acțiunilor (în a patra zonă) rezultate în urma combinării condițiilor;
- se reduce tabela prin fuzionarea regulilor corespunzătoare (şi eliminarea celor redundante dacă este cazul);
- se verifică tabela prin testări succesive, validând versiunea finală.

Pentru o ilustrare sumară a structurii unei tabele de decizie vom prezenta exemplul cu descrierea rezolvării unei ecuații de gradul doi (tratat şi la LNS):

Zona 1: condiții
Identificatorul tabelei: TD1
marcarea regulilor pe verticală: R1 … R4
Zona 3: cu valorile condițiilor

TD1	$(A \cdot X^2+B \cdot X+C=0)$	R1	R2	R3	R4
C1	Coeficientul A nul	DA	DA	NU	NU
C2	Discriminant negativ	DA	NU	DA	NU
A1	Respinge coeficienții	DA	DA		
A2	Determină soluțiile reale				DA
A3	Determină soluțiile complexe			DA	

Zona 2: acțiuni
Zona 4: cu valorile acțiunilor

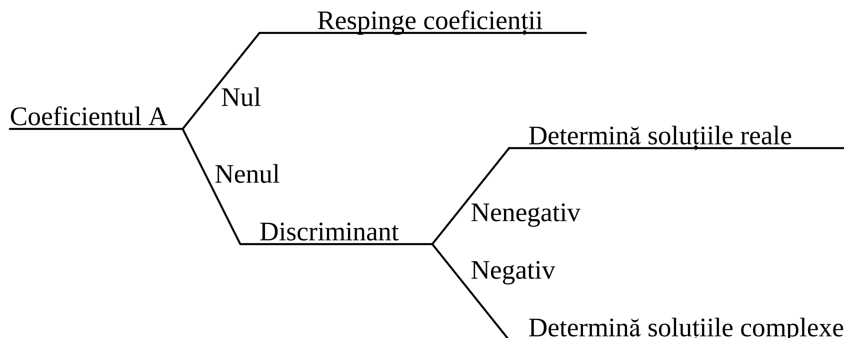
În acest exemplu se poate observa că regulile R1 şi R2 rezultă în acțiuni identice, deci tabela de decizie prezentată (notată TD1) se poate reduce, prin suprapunerea acestor două reguli, obținând o singură regulă din ele (în care valoarea condiției C1 ar fi “DA”, valoarea condiției C2 ar fi “*” adică “ORICE”, iar a acțiunii A1 ar rămâne “DA”).

Dacă o tabelă de decizii conține n condiții independente (în zona 1), numărul de combinații (din zona 3) va fi de 2^n. Dacă fiecare condiție i are un număr de p_i alternative, vor rezulta $p_1 \cdot p_2 \cdot \ldots \cdot p_n$ combinații ale acestor n condiții în total.

2.2.4 Arbori de decizie

Pot fi considerate reprezentări grafice ale unor tabele de decizie, reprezentare în care nodurile arborelor marchează condiții, iar ramurile desemnează acțiuni. Construirea unui arbore de decizie pentru o procedură se poate face în mod asemănător celui de alcătuire a unei tabele de decizii. De multe ori se crează arbori de decizii din tabele de decizii, deoarece arborele (fiind un instrument grafic) constituie un mod mai eficient de comunicare cu utilizatorii.

Iată exemplul rezolvării ecuației de gradul doi, din nou, de data aceasta însă sub forma unui arbore de decizie:



2.2.5 Scheme logice

Sunt cele mai cunoscute instrumente folosite pentru descrierea proceselor, din familia celor grafice. Schema logică este de fapt o reprezentare grafică a paşilor dintr-un algoritm, sub formă de blocuri (simboluri) legate prin linii. Pentru a folosi acest instrument într-o manieră structurată, trebuie să cunoaştem pe lângă primitivele funcționale de bază şi câteva dintre principiile fundamentale, cum ar fi:

- schemele se alcătuiesc şi se citesc de sus în jos (excepțiile se marchează),
- într-un bloc inițial nu intră nici o legătură şi din el pleacă o singură linie de legătură,
- într-un bloc terminal intră oricâte linii de legătură şi nici o linie de legătură nu pleacă din el,
- în toate celelalte blocuri intră cel puțin o linie de legătură, pasul reprezentat de simbol se aplică în cel puțin o succesiune în descrierea algoritmului,
- din blocurile de intrare/ieşire, atribuire şi procedură pleacă o singură linie de legătură,
- dintr-un bloc de decizie pleacă cel puțin două linii de legătură.

Printre avantajele oferite de acest instrument se pot menționa claritatea şi simplitatea reprezentării, restrângerea spațiului necesar prin utilizarea simbolurilor grafice în locul

propozițiilor (față de LNS și pseudocod). Fiind prea analitică, poate ocupa un spațiu destul de mare în cazul prelucrării unui volum mare de date (de exemplu prelucrarea fişierelor), ceea ce este considerat un dezavantaj. Blocurile (simbolurile) folosite la construcția schemelor logice sunt prezentate în tabelul următor.

Tabel cu simbolurile utilizate în scheme logice

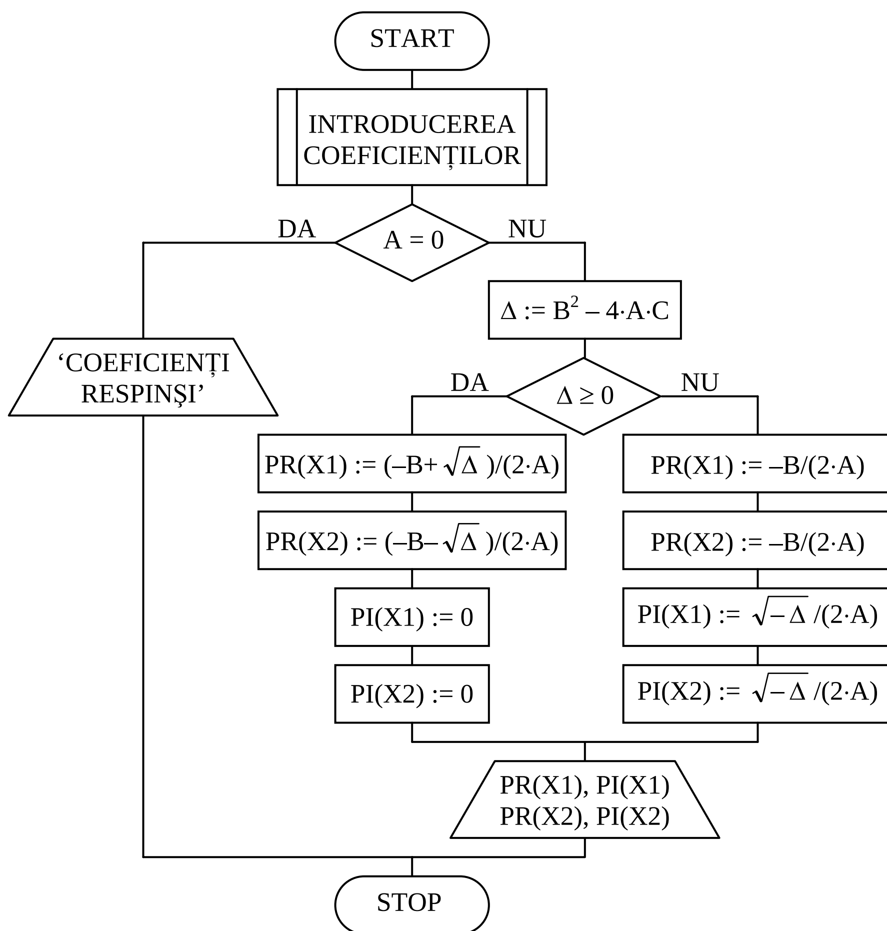
Simbol	Denumire	Utilizare
	Bloc inițial/terminal	Marcarea începutului/sfârşitului schemei
	Bloc de intrare/ieşire	Marcarea operațiilor de citire/scriere
	Bloc de intrare	Marcarea operațiilor de citire
	Bloc de ieşire	Marcarea operațiilor de scriere
	Bloc de atribuire	Punerea în evidență a operațiilor de calcul şi a atribuirii de valori
	Bloc de decizie	Marcarea operațiilor de evaluare prin decidere (apariția ramificațiilor)
	Bloc de procedură sau modul	Marcarea paşilor ce vor fi detaliați ulterior
	Bloc de procedură sau modul (pentru fişiere)	Punerea în evidență a operațiilor de la începutul/sfârşitul prelucrării fişierelor
sau	Linie de legătură	Precizarea modului de înlănțuire a blocurilor (şi marcarea salturilor)
α	Conector intern	Marcarea întreruperii şi continuării unei scheme logice în cadrul aceleiaşi pagini
β	Conector extern	Marcarea întreruperii şi continuării unei scheme logice de pe o pagină pe alta

Primitivele funcționale sunt nişte structuri simple, standardizate în anii 1969–1970 la propunerea lui Dijkstra în scopul structurării programelor. Ele nu sunt specifice doar schemelor logice, dar folosind aceste primitive, construirea şi citirea schemelor logice

devine mai simplă. Dintre principiile utilizării acestor structuri la alcătuirea schemelor logice, menționăm cele mai importante:

- într-o primitivă funcțională intră o singură legătură şi din ea iese o singură legătură,
- un modul dintr-o primitivă funcțională poate conține orice primitivă funcțională.

Pentru exemplificare vom prezenta rezolvarea ecuației de gradul doi (tratată şi la LNS, tabele de decizie şi arbori decizionali):



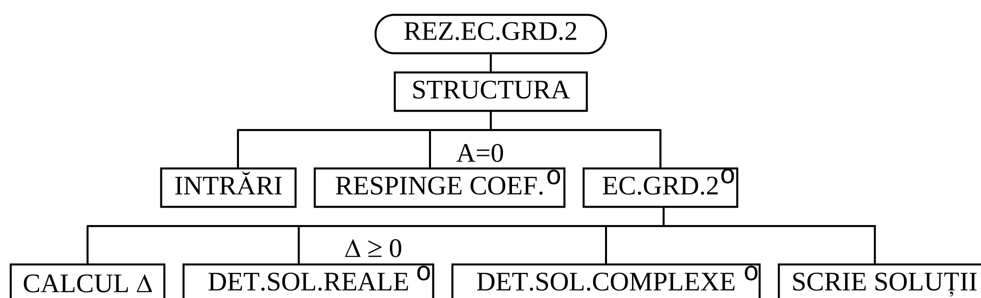
În mod evident nu aceasta este singura modalitate de a rezolva problem propusă. Am ales o singură variantă doar pentru a permite compararea diverselor instrumente descriptive între ele.

2.2.6 Diagrame de structură (de tip Jackson)

Utilizând acest instrument grafic, paşii algoritmului ales sunt reprezentaţi prin module înlănţuite conform unor legi de structură bine precizate. Un modul reprezintă un ansamblu de operaţii cu funcţii bine definite, delimitate fizic prin elemente de început şi de sfârşit, şi care pot fi referite prin nume. Există două tipuri principale de module: module de control (cu rolul de a apela module componente subordonate) şi module funcţionale (care având funcţii concrete, nu se mai descompun). Simbolurile utilizate sunt ilustrate în tabelul următor:

Simbol	Semnificaţie
	Bloc pentru identificarea algoritmului (modul de identificare).
	Bloc pentru un modul executat necondiţionat, secvenţial, o singură dată.
o	Bloc pentru un modul executat cel mult o dată, în funcţie de o condiţie precizată (condiţia se notează deasupra modulului).
*	Bloc pentru un modul executat de mai multe ori, repetitiv, cât timp este îndeplinită condiţia precizată (notată deasupra modulului).

Printre avantajele acestui instrument menţionăm posibilitatea descrierii la niveluri logice diferite în cadrul aceleiaşi scheme, oferind tehnica optimă pentru reprezentarea algoritmilor complecşi. Ca şi dezavantaje se menţionează spaţiul extins ocupat pe orizontală (în funcţie de nivelul de detaliere abordat) şi necesitatea unei descrieri ulterioare destul de laborioase pentru modulele funcţionale (se poate combina cu alte instrumente descriptive). Iată, din nou, rezolvarea ecuaţiei de gradul doi, descrisă prin diagramă de structură (fără detalierea modulelor elementare):

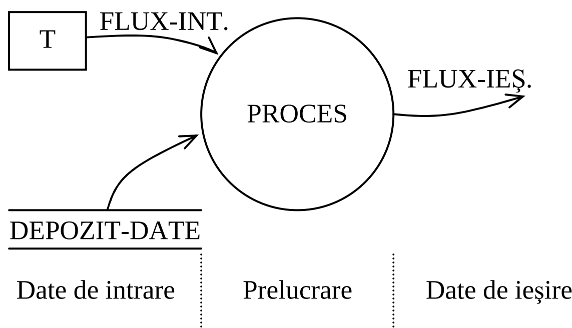


Deşi există încă foarte multe alte instrumente, neprezentate în acest capitol, vom trece în cele ce urmează la prezentarea unei tehnici de bază în modelarea/programarea structurată.

2.2.7 Alte instrumente

În afara celor prezentate există o varietate foarte mare de instrumente, în special cu caracter grafic. Dintre acestea menționăm doar câteva, ele fiind utile mai mult în fazele de analiză şi de proiectare a sistemelor informatice.

Diagramele cu flux de date (DFD) se pot utiliza pentru reprezentarea proceselor atât la nivel fizic cât şi la nivel logic, fiind bazate pe fluxurile de date dintre procese. Ele sunt completate de regulă şi cu alte instrumente, cel puțin cu unele necesare descrierii datelor cum ar fi dicționarul datelor (DD), sau diagrama de structură a datelor (DSD). Avantajul lor constă în principal în schematizarea sugestivă şi modulată oferită pe parcursul etapelor de descompunere (analiză) şi recompunere (proiectare) a sistemelor informatice. Procesele elementare, la rândul lor pot fi descrise prin LNS, tabele de decizie, arbori de decizie etc. Iată cum ar arăta reprezentarea generică a unui program, prin DFD:



Observând maniera de reprezentare oferită de DFD se poate deduce necesitatea ataşării unor descrieri suplimentare. “T” (terminal) şi “DEPOZIT-DATE” sunt periferice standard, marcate ca atare. Dacă prelucrarea marcată prin “PROCES” poate fi detaliată tot printr-un DFD (de nivel mai jos, adică mai descompus), fără un dicționar de date nu putem şti ce înseamnă exact “FLUX-INT.”, “FLUX-IEŞ.”, sau ce se transmite din “DEPOZIT-DATE”. Specificarea corespunzătoare dintr-un dicționar de date ar putea arăta în felul următor:

$$\text{FLUX-INT.} = (\text{COD}) + \text{DESTINAȚIE} + {}^{10}_{0}\{\text{CANTITATE}\} + \begin{bmatrix}\text{JUDEȚ} + \text{LOCALITATE} \\ \text{COD_POSTAL}\end{bmatrix}$$

În acest exemplu “COD” este o componentă opțională, “CANTITATE” se poate repeta de la 0 la 10 ori, iar dintre “JUDEȚ + LOCALITATE” şi “COD_POSTAL” există fie una, fie alta.

2.2.8 Structurile de control primitive (de tip Dijkstra)

Acestea nu reprezintă un instrument în sine, ci aşa cum am menţionat deja la schemele logice, reprezintă o tehnică de structurare a construirii/descrierii algoritmilor, ce poate fi aplicată cu orice instrument de reprezentare. La introducerea lor s-a ţinut cont şi de microspecificaţiile existente în cadrul diferitelor limbaje de programare de nivel înalt, aceste primitive urmărind cerinţele programării structurate prin existenţa în aproape fiecare limbaj de programare a unor instrucţiuni corespunzătoare fiecărei structuri de control primitive.

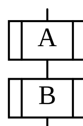
Există trei tipuri de structuri de control primitive: secvenţială, alternativă (selectivă sau decizională) şi repetitivă. Aceste structuri de control precizează de fapt înlănţuirea posibilă a paşilor unui algoritm, conform principiilor programării modulare şi structurate, paşii putând fi: propoziţii, blocuri sau module. În cele ce urmează, vom prezenta cele trei tipuri de structuri de control (prezentând şi câteva subvariante), descriindu-le atât în pseudocod cât şi cu scheme logice şi diagrame de structură Jackson.

1. *Structura secvenţială* (liniară) – apare atunci când orice operaţie se parcurge o dată:

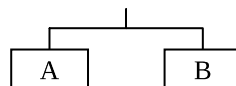
Pseudocod:

DO A
DO B

Schemă logică:



Diagramă de structură (tip Jackson):



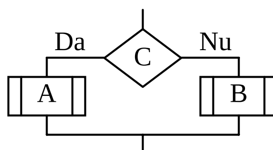
2. *Structurile alternative* (decizionale sau selective) – apar atunci când operaţiile se execută opţional, în funcţie de condiţii precizate:

a. *varianta clasică* (IF-THEN-ELSE):

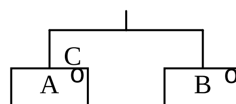
Pseudocod:

```
IF C
 THEN
        DO A
 ELSE
        DO B
ENDIF
```

Schemă logică:



Diagramă de structură (tip Jackson):

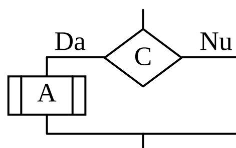


b. *varianta cu ramură vidă* (IF-THEN):

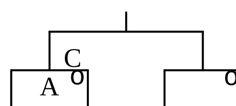
Pseudocod:

IF C DO A

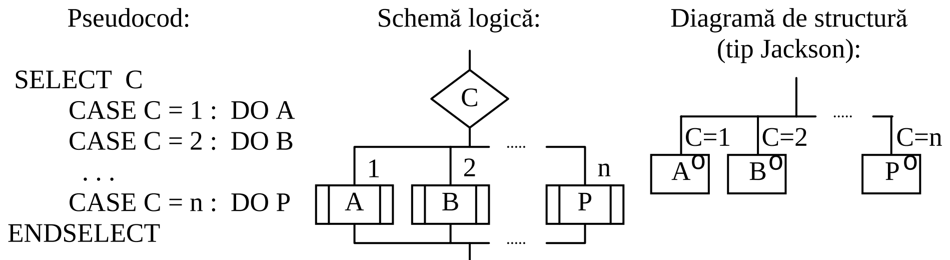
Schemă logică:



Diagramă de structură (tip Jackson):

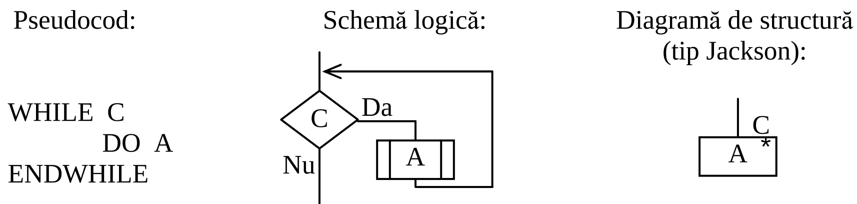


c. *varianta generalizată* (CASE):

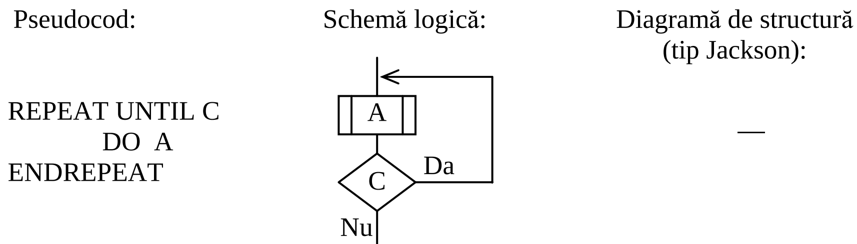


3. *Structurile repetitive* – apar când anumite operații se execută de mai multe ori, în funcție de o condiție precizată:

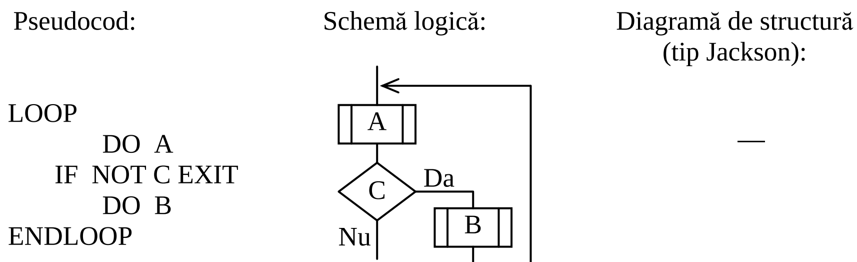
a. *varianta condiționată anterior* (WHILE-DO):



b. *varianta condiționată posterior* (DO-UNTIL):



c. *varianta combinată* (LOOP-EXIT IF-ENDLOOP):



CAPITOLUL 3: FORTRAN 77

3.1 SCRIEREA PROGRAMELOR ÎN LIMBAJUL FORTRAN

În acest capitol vom discuta aspectele legate de redactarea surselor, utilizând limbajul de programare Fortran. Un program scris în acest limbaj poate să conțină una sau mai multe secțiuni (numite uneori module). Secțiunile de program sunt segmente de instrucțiuni şi/sau de date ce pot fi înglobate în unul sau mai multe fişiere. Acestea pot fi compilate şi separat, însă la construirea programului executabil ele vor fi reunite, împreună cu bibliotecile necesare. Pentru redactarea fişierelor ce conțin segmentele de program se poate apela la orice editor de text ce generează fişiere cu conținut afişabil „curat“, însă trebuie avute în vedere şi câteva reguli, prezentate în cele ce urmează.

Setul de caractere constă din caractere alfanumerice (cele 26 litere mici sau mari ale alfabetului englez: a—z, A—Z; şi cifrele: 0—9), 4 simboluri pentru operații aritmetice (adunare: **+**, scădere: **-**, înmulțire: *, împărțire: /, ridicare la putere: **) precum şi dintr-un set determinat de caractere speciale (blank sau spațiu, tabulator orizontal, virgulă, punct, apostrof, paranteze rotunde deschise şi închise, precum şi următoarele caractere: **=**, **$**, **&**). Limbajul Fortran 90 a mai extins această listă cu următoarele caractere speciale admise: **_**, **!**, **:**, **;**, **"**, **%**, <, >, **?**, ^ şi **#**. În general, conform convențiilor anglo-saxone, virgula are rol de separator în cadrul unei liste, iar separatorul zecimal este punctul.

Pentru denumirea diferitelor secțiuni de program precum şi pentru identificarea funcțiilor, variabilelor, tablourilor şi blocurilor se folosesc *nume simbolice*. Dacă convențiile versiunilor mai vechi ale limbajului au permis utilizarea a doar 8 caractere (alcătuite din caractere alfanumerice şi caracterul special **$**), Fortran 90 permite utilizarea a 31 de caractere (alcătuite din caractere alfanumerice, caracterul special **$** şi caracterul special **_**). Primul caracter trebuie să fie întotdeauna o literă. Numele secțiunilor de program sunt considerate *globale* şi trebuie să fie unice în întreaga sursă.

Modul de redactare al sursei poate fi în *format fix* (Fortran 77), *format tabular* sau *format liber* (admise de Fortran 90 şi versiunile ulterioare ale limbajului). Formatul fix respectă structura de redactare bazată pe cartele perforate, considerând lungimea unui rând (articol) de maximum 80 de caractere, având următoarea structură:

Coloane:	1—5	6	7—72	73—80
Conținut:	Etichete. (În prima coloană se poate scrie şi caracterul ce marchează întregul rând explicit ca fiind comentariu).	Caracter ce marchează continuarea rândului anterior (dacă este cazul).	Instrucțiuni.	Comentariu implicit.

Etichetele reprezintă serii de cel mult 5 caractere numerice (cifre) cu rol de referință în cadrul secțiunii de program, ele marcând instrucțiunile în fața cărora apar (în rândul respectiv). Folosirea lor este opțională şi supusă unor restricții (nu toate instrucțiunile pot purta etichetă). Pentru ca o etichetă să fie validă, valoarea ei trebuie să fie cuprinsă în intervalul 1—99999. Dacă se doreşte marcarea rândului curent ca şi comentariu, în prima coloană se va scrie litera **C** sau caracterul ***** (respectiv **!** în cazul versiunilor Fortran 90 şi ulterioare), în acest caz structura şi conținutul rândului fiind ignorate la compilare. Unele compilatoare permit şi folosirea caracterului **D** pentru marcarea în prima coloană a rândului curent ca şi comentariu, această facilitate permițând compilarea (interpretarea) opțională a acestor rânduri în caz de depanare a sursei (*debugging*).

În Fortran 77 se scrie doar o singură instrucțiune într-un rând. Dacă spațiul dintre coloanele 7 şi 72 din rândul curent nu este suficient pentru a scrie instrucțiunea dorită, ea poate fi extinsă marcând în coloana 6 pe următoarele rânduri continuarea celor precedente, prin cifre (doar din intervalul 1—9), litere sau prin unul din caracterele **+**, **-**, ***** (sub Fortran 90 se poate folosi orice caracter în afară de cifra 0). Începând cu versiunea 90 a limbajului se admite scrierea mai multor instrucțiuni pe un rând, în cazul acesta caracterul **;** fiind separatorul dintre instrucțiuni. Numărul liniilor de continuare admise depinde şi de compilatorul ales (Fortran 90 permite până la 90 de linii de continuare în formatul fix şi doar 31 de rânduri de continuare în formatul liber). Unele compilatoare permit extinderea zonei de interpretare a rândurilor până la coloana 80 (chiar coloana 132 în cazul utilizării Fortran 90), dar în mod implicit orice conținut din intervalul coloanelor 72—80 este considerat comentariu şi ca atare ignorat la compilare.

În format liber structura rândurilor din sursă nu conține constrângerile descrise mai sus, instrucțiunile nu se limitează la o anumită încadrare pe coloanele liniilor orice linie putând conține de la 0 la 132 de caractere. În schimb spațiile sunt semnificative, primind rol separator în anumite cazuri, pentru a distinge nume, constante sau etichete de numele, constantele sau etichetele cuvintelor cheie alăturate. Acest format a fost introdus doar începând cu Fortran 90 (acesta acceptă însă şi formatul fix şi tabular). În formatul liber comentariul este indicat de caracterul **!**, iar continuarea unui rând curent prin caracterul **&** la sfârşitul rândului curent (lungimea maximă a unui rând fiind de 132 de caractere). Dacă se scriu mai multe instrucțiuni pe un rând, ele trebuie separate prin caracterul **;** (la sfârşitul unui rând acest caracter se ignoră în mod firesc).

3.2 EXPRESII ÎN FORTRAN

Expresiile sunt alcătuite din *operatori*, *operanzi* şi paranteze. Un operand este o valoare reprezentată printr-o constantă, variabilă, element de tablou sau tablou, sau rezultată din evaluarea unei funcții. Operatorii sunt intrinseci (recunoscuți implicit de compilator şi cu caracter global, deci disponibili întotdeauna tuturor secvențelor de program) sau definiți de utilizator (în cazul în care un operator e descris explicit de programator ca funcție). După modul de operare, putem vorbi de *operatori unari* (ce operează asupra unui singur operand) şi *operatorii binari* (ce operează asupra unei perechi de operanzi).

Orice valoare sau referință la funcție folosită ca operand într-o expresie trebuie să fie definită la momentul evaluării expresiei.

Într-o expresie cu operatori intrinseci având ca operanzi tablouri, aceştia din urmă trebuie să fie compatibili (trebuie să aibă aceeaşi formă), deoarece operatorii specificați se vor aplica elementelor corespondente ale tablourilor, rezultând un tablou corespunzător ca rang şi dimensiune cu operanzii. În cazul în care în expresie pe lângă tablouri există şi un operand scalar, acesta din urmă se va aplica tuturor elementelor de tablou (ca şi cum valoarea scalarului ar fi fost multiplicată pentru a forma un tablou corespunzător). Evaluarea unei expresii are întotdeauna un singur rezultat, ce poate fi folosit pentru atribuire sau ca referință.

Expresiile pot fi clasificate în funcție de natura lor în:

- expresii aritmetice sau numerice,
- expresii de şir (caractere),
- expresii logice.

În variantele mai moderne ale limbajului Fortran (începând cu Fortran 90) există şi expresii considerate ca fiind de *inițializare şi specificare*.

Expresiile numerice, aşa cum sugerează denumirea lor, exprimă calcule numerice, fiind formați din operatori şi operanzi numerici, având rezultat numeric ce trebuie să fie definit matematic (împărțirea la zero, ridicarea unei baze de valoare zero la putere nulă sau negativă, sau ridicarea unei baze de valoare negativă la putere reală constituie operații invalide). Termenul de operand numeric poate include şi valori logice, deoarece acestea pot fi tratate ca întregi într-un context numeric (valoarea logică `.FALSE.` corespunde cu valoarea 0 de tip întreg). Operatorii numerici specifică calculele ce trebuie executate, după cum urmează:

**	ridicare la putere;
*	înmulțire;
/	împărțire (diviziune);
+	adunare sau plus unar (identitate);
–	scădere sau minus unar (negație).

Într-o expresie numerică compusă cu mai mulți operatori, prima dată se vor evalua întotdeauna părțile incluse în paranteze (dinspre interior spre exterior) şi funcțiile, prioritatea de evaluare a operatorilor intrinseci fiind după cum urmează: ridicarea la putere, înmulțirea şi împărțirea, plusul şi minusul unar, adunarea şi scăderea. În cazul operatorilor cu aceeaşi prioritate operațiile vor fi efectuate de la stânga spre dreapta. Prin efect local, operatorii unari pot influența această regulă, generând excepții în cazul unor compilatoare care acceptă asemenea expresii. De exemplu, în cazul expresiei numerice `X**-Y*Z`, deşi ridicarea la putere are prioritate mai mare decât înmulțirea sau negația, evaluarea se va face sub forma $x^{-y \cdot z}$ (pentru forma $x^{-y} \cdot z$ ar fi trebuit să scriem `X**(-Y)*Z`), sau, în cazul

expresiei numerice X/–Y*Z, evaluarea se va face sub forma $\frac{x}{-y \cdot z}$ (pentru $\frac{x}{-y} \cdot z$ ar fi trebuit să scriem X/(–Y)*Z sau –X/Y*Z).

Expresiile sunt *omogene* dacă toţi operanzii sunt de acelaşi tip şi sunt *neomogene* în caz contrar. Tipul valorii rezultate în urma evaluării unei expresii numerice depinde de tipul operanzilor şi de rangul acestora. Dacă operanzii din cadrul expresiei au ranguri diferite, valoarea rezultată va fi de tipul operandului cu cel mai mare rang (cu excepția cazului în care o operație implică o valoare complexă şi una în dublă precizie, rezultatul în asemenea situații fiind de tip complex dublu). La verificarea corectitudinii unei expresii numerice compuse se recomandă să se țină cont şi de tipul valorilor parțiale rezultate în cursul evaluării. Rangul tipurilor de date în ordine descrescătoare este următoarea:

```
(COMPLEX*8)
 COMPLEX*4
(REAL*16)
 REAL*8 şi DOUBLE PRECISION
 REAL*4
(INTEGER*8)
 INTEGER*4
 INTEGER*2
 INTEGER*1
(LOGICAL*8)
 LOGICAL*4
 LOGICAL*2
 LOGICAL*1 şi BYTE
```

Expresiile de şir (caractere) se pot alcătui cu operatorul de concatenare intrinsec + (// în Fortran 90) sau cu funcții create de programator, aplicate asupra unor constante sau variabile de tip caracter. Evaluarea unei asemenea expresii produce o singură valoare de tip caracter. Concatenarea se realizează unind conținuturile de tip caracter de la stânga spre dreapta fără ca eventualele paranteze să influențeze rezultatul. Spațiile conținute de operanzi se vor regăsi şi în rezultat.

Expresiile logice constau din operanzi logici sau numerici combinați cu operatori logici şi/sau relaționali. Rezultatul unei expresii logice este în mod normal o valoare logică (echivalentă cu una din constantele literale logice .TRUE. sau .FALSE.), însă operațiile logice aplicate valorilor întregi vor avea ca rezultat tot valori de tip întreg, ele fiind efectuate bit cu bit în ordinea corespondenței cu reprezentarea internă a acestor valori. Nu se pot efectua operații logice asupra valorilor de tip real (simplă sau dublă precizie), complex sau caracter în mod direct, însă asemenea tipuri de valori pot fi tratate cu ajutorul unor operanzi relaționali în cadrul expresiilor logice. În tabelele următoare vom prezenta operatorii relaționali şi operatorii logici. Cei relaționali au nivel egal de prioritate (se execută de la stânga la dreapta, dar înaintea celor logici şi după cei numerici), iar operatorii

logici sunt dați în ordinea priorității lor la evaluare. Operatorii relaționali sunt binari (se aplică pe doi operanzi), la fel şi operatorii logici, cu excepția operatorului de negație logică (`.NOT.`) care este unar.

Tabel cu operatorii relaționali:

Operator		Semnificație
Fortran 77	Fortran 90	
`.LT.`	`<`	Mai mic decât ... (*Less Than*)
`.LE.`	`<=`	Mai mic sau egal cu ... (*Less or Equal than*)
`.EQ.`	`==`	Egal cu ... (*EQual with*)
`.NE.`	`/=`	Diferit de ... (*Not Equal with*)
`.GT.`	`>`	Mai mare decât ... (*Greater Than*)
`.GE.`	`>=`	Mai mare sau egal cu ... (*Greater or Equal than*)

Tabel cu operatorii logici:

Operator	Semnificație	Prioritate
`.NOT.`	Negație logică (NU), rezultă adevărată dacă operandul are valoarea falsă şi falsă dacă operandul are valoarea adevărată.	mare
`.AND.`	Conjuncție logică (ŞI), rezultă adevărată doar dacă ambii operanzi au valoarea adevărată, în caz contrar rezultă falsă.	mai mică
`.OR.`	Disjuncție logică (SAU), rezultă adevărată dacă unul din operanzi are valoarea adevărată, în caz contrar rezultă falsă.	şi mai mică
`.EQV.`	Echivalență logică, rezultă adevărată dacă ambii operanzi au aceaşi valoare, dacă au valori diferite atunci rezultă falsă.	cea mai mică
`.NEQV.`	Inechivalență logică, rezultă adevărată dacă operanzii sunt diferiți, şi falsă dacă sunt la fel.	
`.XOR.`	Disjuncție logică exclusivă (SAU exclusiv), efect similar cu inechivalența logică.	

Expresiile de inițializare şi specificare pot fi considerate cele care conțin operații intrinseci şi părți constante, respectiv o expresie scalară întreagă. Aşa cum sugerează şi denumirea lor, ele servesc la inițializarea unor valori (de exemplu indicele pentru controlul unui ciclu implicit) sau la specificarea unor caracteristici (de exemplu declararea limitelor de tablouri sau a lungimilor din şiruri de caractere).

Prioritatea de evaluare a operatorilor din cadrul expresiilor neomogene este după cum urmează:

- Operatori unari definiți (funcții);
- Operatori numerici (în următoarea ordine: `**`; `*`, `/`; + unar, – unar; +, –);
- Operatorul de concatenare pentru şiruri (caractere);
- Operatori relaționali (cu prioritate egală: `.EQ.`, `.NE.`, `.LT.`, `.LE.`, `.GT.`, `.GE.`);
- Operatori logici (în ordinea: `.NOT.`; `.AND.`; `.OR.`; `.XOR.`, `.EQV.`, `.NEQV.`).

3.3 INSTRUCȚIUNILE LIMBAJULUI DE PROGRAMARE FORTRAN 77

Prezentarea instrucțiunilor o vom face în ordine alfabetică. Pentru fiecare instrucțiune se va specifica forma generală, semnificația elementelor ce apar în sintaxă, precum şi efectul instrucțiunii. Instrucțiunile executabile pot purta etichete, cele declarative nu.

În ceea ce priveşte notațiile utilizate, vă rugăm să luați în considerare următoarele:

- Instrucțiunile şi cuvintele cheie specifice limbajului de programare sunt scrise cu majuscule îngroșate;
- `[ ]` – parantezele drepte încadrează elemente opționale, aceste paranteze nu fac parte din sintaxa limbajului de programare prezentat;
- `...` – cele trei puncte semnifică repetitivitatea unor elemente în cadrul sintaxei;
- – spațiul (blank-ul) face parte din sintaxă, apare scris ca atare (fără marcaj special);
- `( )` – parantezele rotunde fac parte din sintaxă;
- `,` – virgula face parte din sintaxă, are rol separator în cadrul unei liste;
- `*` – asteriscul face parte din sintaxă, de cele mai multe ori se întâlneşte la instrucțiunile de intrare/ieşire semnificând o valoare implicită (unitatea implicită: consola, sau format implicit: în funcție de natura elementelor din lista de intrare/ieşire).

ACCEPT, citire secvențială cu format:

```
ACCEPT f [,listă]
```

unde: `ACCEPT` – instrucțiunea executabilă;
`f` – referința la format (specificator de format);
`listă` – lista de intrare.

Efect: Citeşte una sau mai multe înregistrări de la consolă, converteşte valorile citite în conformitate cu specificația de format asociată *f*, după care le atribuie elementelor din lista de intrare. Similară cu instrucțiunea `READ`.

Variantă cu format implicit: `ACCEPT *[,listă]`

Atribuire aritmetică/logică/caracter:

```
v=exp
```

unde: `v` – variabilă, element de tablou sau subşir de caractere;
`=` – simbolul pentru operația de atribuire;
`exp` – expresie.

Efect: Atribuie valoarea unei expresii aritmetice, logice sau caracter variabilei *v*. Se recomandă ca tipul variabilei *v* şi tipul expresiei *exp* să corespundă.

ASSIGN, asignare:

```
ASSIGN e TO v
```

unde: `ASSIGN` – instrucţiunea executabilă;
`e` – eticheta unei instrucţiuni executabile sau a unei instrucţiuni FORMAT din aceeaşi unitate de program cu instrucţiunea ASSIGN;
`TO` – cuvânt cheie;
`v` – variabilă întreagă.

Efect: Atribuie eticheta *s* unei variabile întregi *v* pentru a fi utilizată ulterior într-o instrucţiune `GOTO` asignat sau ca un specificator de format în cadrul unei instrucţiuni de citire/scriere.

BACKSPACE, repoziţionare în cadrul unui fişier secvenţial:

```
BACKSPACE u
```

sau

```
BACKSPACE([UNIT=]u[,ERR=e])
```

unde: `BACKSPACE` – instrucţiunea executabilă;
`UNIT` – cuvânt cheie pentru desemnarea unităţii logice;
`u` – expresie întreagă (semnificând numărul unei unităţi logice);
`ERR` – cuvânt cheie pentru tratarea unei erori la executarea instrucţiunii;
`e` – eticheta unei instrucţiuni executabile.

Efect: Repoziţionează un fişier secvenţial, în curs de prelucrare, aflat pe unitatea logică *u*, la începutul înregistrării precedente înregistrării la care s-a făcut accesul prin ultima instrucţiune de intrare/ieşire efectuată înainte de `BACKSPACE`. În cazul unei erori la executarea instrucţiunii se va preda controlul instrucţiunii executabile care poartă eticheta *e* din cadrul aceluiaşi modul de program. Atunci când în sintaxă s-a folosit `ERR=`*e* şi apare o eroare la execuţie, controlul va fi transferat instrucţiunii executabile cu eticheta *e* din cadrul aceleiaşi unităţi de program.

Bloc de date sub formă de subunitate:

```
BLOCK DATA [nume]
```

unde: `BLOCK DATA` – instrucţiunea declarativă;
`nume` – un nume simbolic.

Efect: Specifică un subprogram (modul) executabil ce urmează a fi descris, ca bloc de date.

CALL, apel la subprogram:

```
CALL nume[([p][,p]...)]
```

unde: `CALL` – instrucţiunea executabilă;
`nume` – numele unui subprogram sau al unui punct de intrare (a se vedea instrucţiunile `SUBROUTINE` şi `ENTRY`);
`p` – parametru efectiv (poate fi o expresie sau numele unei variabile, numele unui tablou sau numele unui subprogram).

Efect: Apelează un subprogram sau orice procedură externă având numele *nume*, transferând controlul execuţiei la acesta şi asociind parametrii efectivi *p* parametrilor formali din procedura apelată.

CLOSE, închiderea unui fişier deschis:

```
CLOSE([UNIT=]u[,c=val][,ERR=e])
```

unde: `CLOSE` – instrucţiunea executabilă;
`UNIT` – cuvânt cheie pentru desemnarea unităţii logice;
`u` – expresie întreagă (semnificând numărul unei unităţi logice);
`c` – opţiune, unul din cuvintele cheie: `STATUS`, `DISPOSE` sau `DISP`;
`val` – subşir caracter corespunzător valorii cuvântului cheie *c*, putând fi: `'SAVE'`, `'KEEP'`, `'DELETE'` sau `'PRINT'`.
`ERR` – cuvânt cheie pentru tratarea unei erori la executarea instrucţiunii;
`e` – eticheta unei instrucţiuni executabile.

Efect: Realizează închiderea unui fişier deschis, deconectând fişierul de unitatea logică *u* la care a fost asociat anterior. A se vedea şi instrucţiunea `OPEN`. Atunci când în sintaxă s-a folosit `ERR=`*e* şi apare o eroare la execuţie, controlul va fi transferat instrucţiunii executabile cu eticheta *e* din cadrul aceleiaşi unităţi de program.

COMMON, declararea unei zone comune de memorie:

```
COMMON [/[bc]/] nlist[[,][/[bc]/] nlist]...
```

unde: `COMMON` – instrucţiunea declarativă;
`bc` – numele unui bloc comun;

`nlist` – o listă de nume de variabile, nume de tablouri sau declaratori de tablou, separate prin virgule.

Efect: Permite definirea uneia sau mai multor zone contigue de memorie, numite blocuri comune, având numele specificat şi conținând variabile asociate cu numele blocului.

CONTINUE, continuarea execuției:

```
CONTINUE
```

unde: `CONTINUE` – instrucțiunea executabilă.

Efect: Transferă controlul execuției la următoarea instrucțiune executabilă. Fiind o instrucțiune executabilă, poate purta etichetă, din acest motiv se recomandă utilizarea ei după instrucțiunile declarative ce nu pot fi etichetate, atunci când este cazul.

DATA, definirea datelor sub forma unui modul:

```
DATA nlist/clist/[[,]nlist/clist/]...
```

unde: `DATA` – instrucțiunea declarativă;
`nlist` – este o listă de una sau mai multe variabile, nume de tablouri, elemente de tablouri sau nume de subşiruri caracter, separate prin virgule;
`clist` – este o listă alcătuită din una sau mai multe constante separate prin virgule, de forma:

```
[n*]val[,[n*]val]...
```

unde `n` – este o constantă întreagă fără semn, diferită de zero;
`val` – o valoare constantă.

Efect: Valorile constante din fiecare *clist* sunt atribuite succesiv (în ordine de la stânga la dreapta) câte unei entități specificate în lista *nlist* asociată.

DECODE, citire “internă” cu transformare prin format:

```
DECODE(n,f,var[,ERR=e]) [listă]
```

unde: `DECODE` – instrucțiunea executabilă;
`n` – o expresie întreagă;
`f` – referința la format (specificator de format);

`var` – numele unei variabile, unui tablou, unui element de tablou sau subşir caracter;
`ERR` – cuvânt cheie pentru tratarea unei erori la executarea instrucțiunii;
`e` – eticheta unei instrucțiuni executabile;
`listă` – lista de intrare.

Efect: Citeşte *n* caractere din tamponul *var* şi atribuie valori elementelor din *listă*, valori care au rezultat din conversia conform specificației de format *f*. Atunci când în sintaxă s-a folosit `ERR=`*e* şi apare o eroare la execuție, controlul va fi transferat instrucțiunii executabile cu eticheta *e* din cadrul aceleiaşi unități de program.

DEFINE FILE, definirea structurii unui fişier:

```
DEFINE FILE u(m,n,U,v)[,u(m,n,U,v)]...
```

unde: `DEFINE FILE` – instrucțiunea declarativă;
`u` – variabilă sau constantă întreagă;
`m` – variabilă sau constantă întreagă;
`n` – variabilă sau constantă întreagă;
`U` – variabilă sau constantă întreagă;
`v` – variabilă întreagă.

Efect: Defineşte structura înregistrării unui fişier în acces direct unde *u* este numărul unității logice, *m* este numărul înregistrărilor (de lungime fixă din cauza accesului direct) din fişier, *n* este lungimea în cuvinte a unei înregistrări, *U* este un argument fixat, iar *v* este variabila asociată fişierului (în această variabilă se va memora numărul înregistrării imediat următoare celei curente).

DELETE, ştergerea unei înregistrări:

```
DELETE([UNIT=]u[,REC=r][,ERR=e])
```
sau
```
DELETE(u'r[,ERR=e])
```

unde: `DELETE` – instrucțiunea executabilă;
`UNIT` – cuvânt cheie pentru desemnarea unității logice;
`u` – expresie întreagă (semnificând numărul unei unități logice);
`REC` – cuvânt cheie pentru desemnarea înregistrării;
`r` – expresie întreagă (semnificând numărul înregistrării vizate);
`ERR` – cuvânt cheie pentru tratarea unei erori la executarea instrucțiunii;

`e` – eticheta unei instrucțiuni executabile.

Efect: Şterge înregistrarea specificată prin *r*, din fişierul asociat unității logice *u*, sau cea mai recentă înregistrare accesată. Atunci când în sintaxă s-a folosit `ERR=`*e* şi apare o eroare la execuție, controlul va fi transferat instrucțiunii executabile cu eticheta *e* din cadrul aceleiaşi unități de program.

Declarația de tip:

```
tip var[,var]...
```

unde: tip – este unul din următoarele tipuri de date: `BYTE`, `LOGICAL` (sau `LOGICAL*1`, `LOGICAL*2`, `LOGICAL*4`), `INTEGER` (sau `INTEGER*2`, `INTEGER*4`), `REAL` (sau `REAL*4`, `REAL*8`), `DOUBLE PRECISION`, `COMPLEX` (sau `COMPLEX*8`), `CHARACTER` (sau `CHARACTER*`lungime);

`var` – numele unei variabile, unui tablou, unei funcții externe, unei funcții aritmetic definite, unei funcții parametru formal sau al unui declarator de tablou. Numele poate fi urmat opțional de un specificator de lungime de forma: `*n`

unde: `n` – este o expresie întreagă semnificând lungimea lui *var* în octeți (în cazul entităților caracter semnifică numărul de caractere).

Efect: Numele simbolic *var* va avea asignat tipul specificat prin *tip*. Fiind declarație, nu poate purta etichetă. Declarația de tip poate fi combinată cu declararea dimensiunilor pentru tablouri, în acest caz nemaifiind necesară utilizarea instrucțiunii `DIMENSION` într-un mod explicit (în asemenea cazuri *var* va avea forma: `a(d)` – cu semnificația termenilor de la declarația `DIMENSION`.

DIMENSION, declararea dimensiunilor:

```
DIMENSION a(d)[,a(d)]...
```

unde: `DIMENSION` – instrucțiune declarativă;

`a` – numele tabloului;

`d` – declaratorul de dimensiune, sub forma: `n[,n]...`

unde: `n` – este o expresie întreagă (semnificând numărul maxim de elemente în dimensiunea respectivă).

Efect: Specifică spațiul de memorie necesar tablourilor.

DO, instrucțiune pentru cicluri repetitive:

```
DO e[,] c=i,f[,p]
```

unde: DO – instrucțiune executabilă;
e – eticheta unei instrucțiuni executabile;
c – variabilă (variabila de control al ciclului);
i – expresie numerică (desemnând o valoare inițială);
f – expresie numerică (desemnând o valoare finală);
p – expresie numerică (desemnând pasul variabilei de control).

Efect: Execută ciclul DO (instrucțiunile ce urmează, până la cea care poartă eticheta *e* inclusiv, acestea alcătuind corpul ciclului), realizând următoarele faze:

1. Evaluează: `cnt=INT((f-i+p)/p)` (*cnt* fiind contorul ciclului).
2. Execută atribuirea: `c=i`
3. Dacă *c* este mai mic sau egal cu zero, nu se va executa ciclul.
4. Dacă *c* este mai mare ca zero, atunci:
   a. Execută instrucțiunile din corpul ciclului.
   b. Evaluează: `c=c+p`
   c. Decrementează contorul ciclului: `cnt=cnt-1` şi dacă *cnt* este mai mare decât zero, repetă ciclul.

Varianta de ciclu implicit: `([listă,] c=i,f[,p])`

unde: listă – listă de intrare/ieşire;
celelalte valori având semnificațiile de la instrucțiunea DO.

Efect: Determină executarea ciclului asupra elementelor din *listă* în cadrul unei operații de intrare/ieşire. Condițiile de realizare ale ciclului sunt similare cu cele de la instrucțiunea DO.

Notă: Varianta de ciclu implicit poate fi folosită doar în cadrul listelor de intrare/ieşire de la instrucțiunile de citire/scriere. Această variantă nu reprezintă o instrucțiune de sine stătătoare, este considerată doar expresie de inițializare.

ENCODE, scriere "internă" cu transformare prin format:

```
ENCODE(n,f,var[,ERR=e]) [listă]
```

unde: ENCODE – instrucțiunea executabilă;

`n` – expresie întreagă;
`f` – referința la format (specificator de format);
`var` – numele unei variabile, unui tablou, unui element de tablou sau subşir caracter;
`ERR` – cuvânt cheie pentru tratarea unei erori la executarea instrucțiunii;
`e` – eticheta unei instrucțiuni executabile;
`listă` – lista de ieşire.

Efect: Scrie *n* caractere din *listă* în tamponul *var*, care va primi caracterele convertite conform specificației de format *f*. Atunci când în sintaxă s-a folosit `ERR=`*e* şi apare o eroare la execuție, controlul va fi transferat instrucțiunii executabile cu eticheta *e* din cadrul aceleiaşi unități de program.

END, marcaj de sfârşit:

```
END
```

unde: `END` – instrucțiunea declarativă;

Efect: Marchează sfârşitul unei unități de program (fiind obligatoriu pentru compilare).

ENDFILE, scrierea unei înregistrări EOF:

```
ENDFILE u
```
sau
```
ENDFILE([UNIT=]u[,ERR=e])
```

unde: `ENDFILE` – instrucțiunea executabilă;
`UNIT` – cuvânt cheie pentru desemnarea unității logice;
`u` – expresie întreagă (semnificând numărul unei unități logice);
`ERR` – cuvânt cheie pentru tratarea unei erori la executarea instrucțiunii;
`e` – eticheta unei instrucțiuni executabile.

Efect: Scrierea unei înregistrări de sfârşit de fişier (`EOF` – *End Of File*) în fişierul secvențial asociat unității logice *u*. Atunci când în sintaxă s-a folosit `ERR=`*e* şi apare o eroare la execuție, controlul va fi transferat instrucțiunii executabile cu eticheta *e* din cadrul aceleiaşi unități de program.

ENTRY, declararea unui punct de intrare:

```
ENTRY nume[(p[,p]...)]
```

unde: ENTRY – instrucțiunea declarativă;
nume – numele punctului de intrare;
p – un nume simbolic reprezentând un parametru formal.

Efect: Permite crearea unor puncte de intrare multiple în unitățile de program declarate ca FUNCTION şi SUBROUTINE.

EQUIVALENCE, echivalare prin declarare:

```
EQUIVALENCE (nlist)[,(nlist)]...
```

unde: EQUIVALENCE – instrucțiunea declarativă;
nlist – este o listă de cel puțin două variabile, nume de tablouri, elemente de tablouri sau subşiruri caracter, separate prin virgule. Expresiile de indici trebuie să fie constante întregi.

Efect: Alocă fiecărei entități din *nlist* aceeaşi locație de memorie.

EXTERNAL, declararea unor module externe:

```
EXTERNAL nume[,nume]...
```
sau
```
EXTERNAL *nume[,*nume]...
```

unde: EXTERNAL – instrucțiunea declarativă;
nume – numele unei unități de program.

Efect: Defineşte numele specificat ca fiind numele unei unități de program. Când *nume* este precedat de *****, defineşte o unitate (subprogram) externă furnizată de utilizator.

FIND, poziționare în fişier:

```
FIND(u'r[,ERR=e])
```
sau
```
FIND([UNIT=]u[,REC=r][,ERR=e])
```

unde: `FIND` – instrucţiunea executabilă;
`UNIT` – cuvânt cheie pentru desemnarea unităţii logice;
`u` – expresie întreagă (semnificând numărul unei unităţi logice);
`REC` – cuvânt cheie pentru desemnarea înregistrării;
`r` – expresie întreagă (semnificând numărul înregistrării vizate);
`ERR` – cuvânt cheie pentru tratarea unei erori la executarea instrucţiunii;
`e` – eticheta unei instrucţiuni executabile.

Efect: Poziţionează fişierul în acces direct de pe unitatea logică *u* pe articolul specificat prin *r*. Atunci când în sintaxă s-a folosit `ERR=`*e* şi apare o eroare la execuţie, controlul va fi transferat instrucţiunii executabile cu eticheta *e* din cadrul aceleiaşi unităţi de program.

FORMAT, declararea formei la operaţii de intrare/ieşire:

```
FORMAT(listă)
```

unde: `FORMAT` – instrucţiunea de specificare;
`listă` – listă de una sau mai mulţi descriptori (specificatori de câmp).

Efect: Descrie formatul în care urmează să se transmită, prin intermediul instrucţiunilor de citire/scriere, una sau mai multe înregistrări. Instrucţiunea trebuie întotdeauna etichetată, altfel nu-şi are rostul.

Funcţie aritmetic definită:

```
nume([p[,p]...])=exp
```

unde: `nume` – nume simbolic (denumirea funcţiei);
`p` – nume simbolic (al parametrului formal);
`exp` – expresie.

Efect: Permite realizarea unei proceduri definite printr-o singură instrucţiune şi având *p* drept parametru formal. Când funcţia *nume* astfel definită este referită, se evaluează expresia *exp* utilizând parametri efectivi de apel. Este o instrucţiune declarativă ce poate fi precedată de o declaraţie de tip.

FUNCTION, declararea unei funcții ca modul:

```
FUNCTION nume[([p[,p]...])]
```

unde: `FUNCTION` – instrucțiunea declarativă;
`nume` – nume simbolic (asociat modulului);
`p` – nume simbolic (parametru formal).

Efect: Defineşte o unitate de program ca funcție externă (modul separat de program) având numele indicat prin *nume* şi utilizând parametri formali *p*. Transferarea valorilor parametrilor se efectuează prin corespondența celor formali *p* (ca ordine şi tip) cu cei efectivi, la invocarea funcției prin *nume*.

GO TO, salt necondiționat:

```
GO TO e
```
sau
```
GOTO e
```

unde: `GO TO` – cuvinte cheie ale instrucțiunii executabile;
`e` – eticheta unei instrucțiuni executabile.

Efect: Determină transferul controlului la executarea instrucțiunii cu eticheta *e* din cadrul aceleiaşi unități de program.

GO TO, salt calculat:

```
GO TO (listă)[,]exp
```
sau
```
GOTO (listă)[,]exp
```

unde: `GO TO` – cuvinte cheie ale instrucțiunii executabile;
`listă` – listă de una sau mai multe etichete de instrucțiuni executabile, separate prin virgulă;
`exp` – expresie aritmetică (întreagă).

Efect: Transferă controlul la instrucțiunea a cărei etichetă ocupă poziția *exp* în *listă*. Dacă *exp* este mai mic decât 1 sau mai mare decât numărul etichetelor din *listă*, nu se va efectua transferul.

GO TO, salt asignat:

```
GO TO v[[,](listă)]
      sau
GOTO v[[,](listă)]
```

unde: `GO TO` – cuvinte cheie ale instrucțiunii executabile;
`v` – variabilă de tip întreg;
`listă` – listă de una sau mai multe etichete de instrucțiuni executabile, separate prin virgulă.

Efect: Transferă controlul la instrucțiunea a cărei etichetă a fost atribuită variabilei *v* printr-o instrucțiune prealabilă `ASSIGN`. Dacă se utilizează *listă*, valoarea atribuită lui *v* trebuie să aparțină listei (în caz contrar, nu are loc transferul).

IF aritmetic:

```
IF(exp)e1,e2,e3
```

unde: `IF` – instrucțiunea executabilă;
`exp` – expresie aritmetică;
`e1`, `e2`, `e3` – etichetele unor instrucțiuni executabile.

Efect: Transferă controlul la instrucțiunea cu eticheta *e1*, *e2* sau *e3* din aceeași unitate de program, în funcție de valoarea expresiei *exp*: dacă *exp* rezultă mai mic ca zero, controlul este transferat la instrucțiunea cu eticheta *e1*; dacă *exp* rezultă zero, controlul este transferat instrucțiunii cu eticheta *e2*; dacă *exp* rezultă mai mare ca zero controlul este transferat instrucțiunii cu eticheta *e3*.

IF logic simplu:

```
IF(exp) inst
```

unde: `IF` – instrucțiunea executabilă;
`exp` – expresie logică;
`inst` – orice instrucțiune executabilă cu excepția următoarelor: `DO`, `END`, `IF` logic simplu, `IF` logic structurat (IF-THEN sau bloc IF).

Efect: Execută instrucțiunea *inst* dacă expresia logică *exp* are valoare `.TRUE.` (adevărat). În caz contrar, se execută instrucțiunea care urmează instrucțiunii `IF` logic fără a se mai executa instrucțiunea *inst*.

IF logic structurat (blocul IF):

```
IF(exp1) THEN
      [bloc]
[ELSE [IF(exp2) THEN
      [bloc]
[ELSE
      bloc]]
ENDIF
```

unde: `IF` – instrucțiunea executabilă;
`exp1` – expresie logică;
`exp2` – expresie logică;
`THEN` – cuvânt cheie (obligatoriu);
`ELSE` – cuvânt cheie (opțional); defineşte un bloc de instrucțiuni ce urmează a fi executate dacă expresiile logice din instrucțiunile IF-THEN precedente au valoarea `.FALSE.` (falsă);
`bloc` – o secvență de una sau mai multe instrucțiuni;
`ENDIF` – cuvânt cheie (obligatoriu); marchează terminarea unui bloc `IF`.

Efect: Defineşte blocuri de instrucțiuni şi le execută condiționat. Dacă expresia logică *exp1* din instrucțiunea IF-THEN are valoarea `.TRUE.`, se va executa primul *bloc* şi controlul se va transfera la prima instrucțiune executabilă după cuvântul cheie `ENDIF`. Dacă expresia logică *exp1* are valoarea logică `.FALSE.`, procedura se va repeta pentru următoarea instrucțiune `ELSE` IF-THEN. Dacă toate expresiile logice au valoarea `.FALSE.`, se va executa *bloc*-ul ce urmează cuvântului cheie `ELSE`. Dacă acest *bloc* nu există, controlul se va transfera la următoarea instrucțiune executabilă care urmează cuvântului cheie `ENDIF`.

IMPLICIT, declararea naturii implicite de tip:

```
IMPLICIT tip (a[,a]...)[,tip (a[,a]...)]...
```

unde: `IMPLICIT` – instrucțiunea declarativă;
`tip` – specificator de tip (a se vedea Declarația de tip);
`a` – fie o singură literă, fie două litere, în ordine alfabetică, separate printr-o liniuță (de exemplu: `A,D–F`).

Efect: Atribuie tipul specificat tuturor entităților al căror nume simbolic începe cu una din literele aparținând domeniului descris între paranteze.

INCLUDE, inserare de cod:

```
INCLUDE 'specfis'[ /opt]
```

unde: `INCLUDE` – instrucțiunea declarativă;
`specfis` – un specificator de fişier (citat);
`opt` – opțiune (comutator opțional) cu una din următoarele două forme:

`LISTF` – se listează instrucțiunile incluse (implicit);
`NOLIST` – nu se listează instrucțiunile din fişierul inclus.

Efect: Include instrucțiunile sursă din fişierul specificat prin *specfis* în compilarea fişierului sursă curent.

INTRINSIC, redeclararea funcțiilor interne:

```
INTRINSIC nume[,nume]...
```

unde: `INTRINSIC` – instrucțiunea declarativă;
`nume` – numele simbolic al unei funcții intrinseci (interne).

Efect: Desemnează numele simbolic ca funcții intrinseci şi permite utilizarea acestor nume în cadrul unității curente de program, cu parametri efectivi. Se subînțelege că aceste funcții trebuie să fie funcții existente, predefinite intern.

OPEN, deschiderea fişierelor:

```
OPEN(p[,p]...)
```

unde: `OPEN` – instrucțiunea executabilă;
`p` – parametru, fiind specificația unui cuvânt cheie, de forma:

`cuv` sau `cuv=val`

unde `cuv` – este cuvânt cheie (a se vedea tabelul următor);
`val` – valoare în funcție de cuvântul cheie *cuv* (a se vedea tabelul următor).

Efect: Deschide un fişier asociindu-l cu unitatea logică *u* specificată, în conformitate cu parametrii specificați prin cuvintele cheie.

Tabel cu parametrii din instrucțiunea OPEN (în ordine alfabetică):

Cuvântul cheie (*cuv*)	Valoarea (*val*)	Funcțiune	Implicit
ACCESS	'SEQUENTIAL' 'DIRECT' 'APPEND' 'KEYED'	Metoda de acces	'SEQUENTIAL'
ASSOCIATEVARIABLE	*val*	Numărul înregistrării următoare în accesul direct	Nu există variabilă asociată
BLANK	'NULL' 'ZERO'	Interpretarea spațiilor (blank-urilor)	'NULL'
BLOCKSIZE	*val*	Dimensiunea tamponului de intrare/ieşire	Alocată de sistem
BUFFERCOUNT	*val*	Numărul de tampoane de intrare/ieşire	Alocat de sistem
CARRIAGECONTROL	'FORTRAN' 'LIST' 'NONE'	Controlul (interpretarea) returului de car	'FORTRAN' în cazul formatat, şi 'NONE' în cazul neformatat
DISPOSE sau DISP	'SAVE' 'KEEP' 'PRINT' 'DELETE'	Starea fişierului la închidere	'SAVE'
ERR	*e*	Eticheta de transfer la eroare	Nu se face transfer la eroare
EXTENDSIZE	*val*	Extensie de alocare a spațiului de memorie pentru fişier	Dată de sistemul de operare sau de volum (partiție)
FILE sau NAME	*specfis*	Specificator de fişier	Depinde de unitate şi de sistem
FORM	'FORMATTED' 'UNFORMATTED'	Formatul fişierului	Depinde de cuvântul cheie ACCESS
INITIALSIZE	*val*	Spațiu de memorie alocat pentru fişier	Nu se alocă
KEY	(k[,k]...)	Câmpurile de cheie pentru fişier indexat	Nu este implicit

Cuvântul cheie (*cuv*)	Valoarea (*val*)	Funcțiune	Implicit
`MAXREC`	*val*	Numărul maxim de înregistrări în accesul direct	Nu există maximum
`NOSPANBLOCKS`		Înregistrările nu traversează blocurile	Înregistrările pot traversa blocurile
`ORGANIZATION`	`'SEQUENTIAL'` `'RELATIVE'` `'INDEXED'`	Structura fişierului	`'SEQUENTIAL'`
`READONLY`		Protecție la scriere	Neprotejat la scriere
`RECL` sau `RECORDSIZE`	*val*	Lungimea înregistrării	Depinde de cuvintele cheie: `TYPE`, `ORGANIZATION`, `RECORDTYPE`
`RECORDTYPE`	`'FIXED'` `'VARIABLE'` `'SEGMENTED'`	Structura înregistrării	Depinde de cuvintele cheie: `ACCESS`, `FORM`
`SHARED`		Acces partajat la fişier	Nu este permis accesul partajat la fişier
`STATUS` sau `TYPE`	`'OLD'` `'NEW'` `'SCRATCH'` `'UNKNOWN'`	Starea fişierului la deschidere	`'UNKNOWN'`
`UNIT`	*u*	Numărul unității logice asociate fişierului	Nu este implicit
`USEROPEN`	*nume*	Opțiune pentru un program utilizator	Nu există opțiune

Semnificația notațiilor din coloana valorilor *val* (exceptând cuvintele cheie):

val – valoare numerică;
e – eticheta unei instrucțiuni executabile;
specfis – specificator de fişier;
k – numele simbolic asociat câmpului de cheie;
u – numărul unității logice;
nume – numele simbolic al unei unități de program.

PARAMETER, declarație pentru parametri:

```
PARAMETER (p=c[,p=c]...)
```

unde: `PARAMETER` – instrucțiunea declarativă;
`p` – nume simbolic;
`c` – constantă.

Efect: Defineşte un nume simbolic pentru o constantă.

PAUSE, suspendarea temporară a execuției:

```
PAUSE [şir]
```

unde: `PAUSE` – instrucțiunea executabilă;
`şir` – un şir constituit din maximum cinci caractere numerice (cifre), o constantă octală sau un literal alfanumeric.

Efect: Suspendă execuția programului şi afişează *şir* la terminal. Execuția programului se continuă doar după comanda dată de la tastatură de către utilizator.

PRINT, scriere secvențială:

```
PRINT f[,listă]
```

unde: `PRINT` – instrucțiunea executabilă;
`f` – referință la format (specificator de format);
`listă` – listă de ieşire.

Efect: Scrie o înregistrare pe dispozitivul de ieşire implicit (consolă/monitor sau imprimantă), conținând valorile elementelor din *listă*. Valorile sunt convertite în conformitate cu specificația de format *f*.

Varianta cu format implicit: `PRINT *[,listă]`

PROGRAM, definirea programului principal:

```
PROGRAM nume
```

unde: `PROGRAM` – instrucțiunea declarativă;

`nume` – nume simbolic (asociat corpului programului principal).

Efect: Specifică un nume pentru programul principal. Declarația fiind opțională, în cazul în care este omisă, programul principal va purta numele simbolic `MAIN`.

READ, citire secvențială:

```
READ([UNIT=]u[,[FMT=]f][,END=e1][,ERR=e2]) [listă]
```
sau
```
READ  f[,listă]
```

unde: `READ` – instrucțiunea executabilă;
`UNIT` – cuvânt cheie pentru desemnarea unității logice;
`u` – expresie întreagă (semnificând numărul unei unități logice);
`FMT` – cuvânt cheie pentru referința de format;
`f` – referință la format (specificator de format);
`END` – cuvânt cheie pentru tratarea întâlnirii sfârşitului fişierului (EOF);
`e1` – eticheta unei instrucțiuni executabile;
`ERR` – cuvânt cheie pentru tratarea unei erori la executarea instrucțiunii;
`e2` – eticheta unei instrucțiuni executabile;
`listă` – listă de intrare (cu elemente separate prin virgulă).

Efect: Citeşte una sau mai multe înregistrări logice de la unitatea *u* şi atribuie valori elementelor din *listă*. Valorile sunt convertite în conformitate cu specificatorul de format *f*. Atunci când în sintaxă s-a folosit `END=`*e1* şi la executarea instrucțiunii se întâlneşte sfârşitul fişierului (`EOF`), controlul va fi transferat instrucțiunii executabile cu eticheta *e1* din cadrul aceleiaşi unități de program. Atunci când în sintaxă s-a folosit `ERR=`*e2* şi apare o eroare la execuție, controlul va fi transferat instrucțiunii executabile cu eticheta *e2* din cadrul aceleiaşi unități de program.

Varianta cu format implicit:

```
READ([UNIT=]u,[FMT=]*[,END=e1][,ERR=e2]) [listă]
```
sau
```
READ  *[,listă]
```

Efect: Citeşte una sau mai multe înregistrări logice de la unitatea *u* şi atribuie valori elementelor din *listă*. Valorile sunt convertite în conformitate cu tipul elementelor din *listă*.

Varianta fără format:

```
READ([UNIT=]u[,END=e1][,ERR=e2]) [listă]
```

Efect: Citeşte o înregistrare (un articol) fără format de la unitatea logică *u* şi atribuie valori elementelor din *listă*.

READ, citire în acces direct:

```
READ([UNIT=]u,REC=r[,[FMT=]f][,ERR=e]) [listă]
```
sau
```
READ(u'r[,[FMT=]f][,ERR=e]) [listă]
```

unde: `READ` – instrucţiunea executabilă;
`UNIT` – cuvânt cheie pentru desemnarea unităţii logice;
`u` – expresie întreagă (semnificând numărul unei unităţi logice);
`REC` – cuvânt cheie pentru desemnarea înregistrării;
`r` – expresie întreagă (semnificând numărul înregistrării vizate);
`FMT` – cuvânt cheie pentru referinţa de format;
`f` – referinţă la format (specificator de format);
`ERR` – cuvânt cheie pentru tratarea unei erori la executarea instrucţiunii;
`e` – eticheta unei instrucţiuni executabile;
`listă` – listă de intrare (cu elemente separate prin virgulă).

Efect: Citeşte înregistrări pornind cu înregistrarea *r* de la unitatea logică *u* şi atribuie valori elementelor din *listă*. Valorile sunt convertite în conformitate cu specificatorul de format *f*. Atunci când în sintaxă s-a folosit `ERR=`*e* şi apare o eroare la execuţie, controlul va fi transferat instrucţiunii executabile cu eticheta *e* din cadrul aceleiaşi unităţi de program.

Varianta fără format:

```
READ([UNIT=]u,REC=r[,ERR=e]) [listă]
```
sau
```
READ(u'r[,ERR=e]) [listă]
```

Efect: Citeşte înregistrarea *r* de la unitatea logică *u* şi atribuie valori elementelor din *listă*.

Notă: Citirea directă se aplică fişierelor cu organizare relativă. Un fişier secvenţial poate fi citit în acces direct numai dacă înregistrările au lungimi egale (fixe) şi fişierul a fost asociat cu o unitate logică prin instrucţiunea `OPEN` utilizând şi parametrul `ACCESS='DIRECT'`. A se vedea instrucţiunea `OPEN`.

READ, citire indexată:

```
READ([UNIT=]u[,[FMT=]f],cc=vc[,KEYID=nc][,ERR=e]) [listă]
```

unde: `READ` – instrucțiunea executabilă;
`UNIT` – cuvânt cheie pentru desemnarea unității logice;
`u` – expresie întreagă (semnificând numărul unei unități logice);
`FMT` – cuvânt cheie pentru referința de format;
`f` – referință la format (specificator de format);
`cc` – cuvânt cheie, una din următoarele:

`KEY`, `KEYEQ`, `KEYGE` sau `KEYGT`;

`vc` – expresie (valoare) de cheie;
`KEYID` – cuvânt cheie pentru referința cheii;
`nc` – expresie întreagă (numărul de referință sau rangul cheii);
`ERR` – cuvânt cheie pentru tratarea unei erori la executarea instrucțiunii;
`e` – eticheta unei instrucțiuni executabile;
`listă` – listă de intrare (cu elemente separate prin virgulă).

Efect: Citeşte înregistrarea, de la unitatea logică *u*, descrisă de expresia de cheie *vc* şi numărul de referință al cheii *nc*. Valorile din înregistrare sunt convertite în conformitate cu specificatorul de format *f* şi vor fi atribuite elementelor din *listă*. Dacă cuvântul cheie `KEYID` este omis (şi implicit referința de cheie *nc*), atunci se subînțelege un singur câmp definit ca şi cheie în structura înregistrării. Atunci când în sintaxă s-a folosit `ERR=`*e* şi apare o eroare la execuție, controlul va fi transferat instrucțiunii executabile cu eticheta *e* din cadrul aceleiaşi unități de program.

Varianta fără format:

```
READ([UNIT=]u,cc=vc[,KEYID=nc][,ERR=e]) [listă]
```

Efect: Citeşte înregistrarea, de la unitatea logică *u*, descrisă de expresia de cheie *vc* şi numărul de referință al cheii *nc*. Valorile din înregistrare vor fi atribuite elementelor din *listă*.

Notă: Citirea indexată se poate aplica doar fişierelor indexate care conțin una sau mai multe câmpuri de înregistrare declarate sub formă de cheie. A se vedea instrucțiunea `OPEN`.

READ, citire internă:

```
READ([UNIT=]c,[FMT=]f[,END=e1][,ERR=e2]) [listă]
```

unde: `READ` – instrucțiunea executabilă;
`UNIT` – cuvânt cheie pentru desemnarea unității logice;
`c` – specificator de fişier intern;
`FMT` – cuvânt cheie pentru referința de format;
`f` – referință la format (specificator de format);
`END` – cuvânt cheie pentru tratarea întâlnirii sfârşitului fişierului (EOF);
`e1` – eticheta unei instrucțiuni executabile;
`ERR` – cuvânt cheie pentru tratarea unei erori la executarea instrucțiunii;
`e2` – eticheta unei instrucțiuni executabile;
`listă` – listă de intrare (elementele separate prin virgulă).

Efect: Citeşte date caracter de la un fişier intern *c*, transformă datele din caracter în formă binară utilizând specificatorul de format *f*, atribuie datele transformate elementelor din *listă* în ordine, de la stânga la dreapta. Atunci când în sintaxă s-a folosit `END=`*e1* şi la executarea instrucțiunii se întâlneşte sfârşitul fişierului (`EOF`), controlul va fi transferat instrucțiunii executabile cu eticheta *e1* din cadrul aceleiaşi unități de program. Atunci când în sintaxă s-a folosit `ERR=`*e2* şi apare o eroare la execuție, controlul va fi transferat instrucțiunii executabile cu eticheta *e2* din cadrul aceleiaşi unități de program.

Notă: Citirea internă se utilizează pentru convertirea caracterelor. A se vedea şi instrucțiunea `DECODE`.

RETURN, revenire din subunitatea curentă de program în cel apelant:

```
RETURN
```

unde: `RETURN` – intrucțiune executabilă.

Efect: Întoarce controlul programului apelant din cadrul subprogramului curent.

REWIND, repoziționare:

```
REWIND([UNIT=]u[,ERR=e])
```
sau
```
REWIND u
```

unde: `REWIND` – instrucțiunea executabilă;
`UNIT` – cuvânt cheie pentru desemnarea unității logice;
`u` – expresie întreagă (semnificând numărul unei unități logice);

`ERR` – cuvânt cheie pentru tratarea unei erori la executarea instrucțiunii;
`e` – eticheta unei instrucțiuni executabile.

Efect: Repoziționează unitatea logică *u* la începutul fişierului curent, deschis. Atunci când în sintaxă s-a folosit `ERR=`*e* şi apare o eroare la execuție, controlul va fi transferat instrucțiunii executabile cu eticheta *e* din cadrul aceleiaşi unități de program.

REWRITE, rescriere:

```
REWRITE([UNIT=]u[,[FMT=]f][,ERR=e]) [listă]
```

unde: `REWRITE` – instrucțiunea executabilă;
`UNIT` – cuvânt cheie pentru desemnarea unității logice;
`u` – expresie întreagă (semnificând numărul unei unități logice);
`FMT` – cuvânt cheie pentru referința de format;
`f` – referință la format (specificator de format);
`ERR` – cuvânt cheie pentru tratarea unei erori la executarea instrucțiunii;
`e` – eticheta unei instrucțiuni executabile;
`listă` – lista de ieşire.

Efect: Rescrie înregistrarea curentă de pe unitatea logică *u*, utilizând valorile elementelor din *listă*. Valorile sunt convertite conform referinței de format *f* (dacă aceasta a fost specificată). Atunci când în sintaxă s-a folosit `ERR=`*e* şi apare o eroare la execuție, controlul va fi transferat instrucțiunii executabile cu eticheta *e* din cadrul aceleiaşi unități de program.

SAVE, salvarea entităților:

```
SAVE [a[,a]...]
```

unde: `SAVE` – instrucțiunea declarativă;
`a` – numele unui bloc comun (incadrat de /-uri), nume de variabilă sau de tablou.

Efect: Păstrează definirea curentă a entităților *a* după întâlnirea unei instrucțiuni `END` sau `RETURN` a unui subprogram.

STOP, oprirea execuției:

```
STOP [şir]
```

unde: `STOP` – instrucțiunea executabilă;
`şir` – un şir de maximum cinci cifre zecimale, un literal alfanumeric sau o constantă octală.

Efect: Termină execuția programului, afişând la terminalul utilizatorului *şir*-ul specificat.

SUBROUTINE, declararea unui subprogram:

```
SUBROUTINE nume[([p[,p]...])]
```

unde: `SUBROUTINE` – instrucțiunea declarativă;
`nume` – nume simbolic (asociat subprogramului);
`p` – nume simbolic (parametru formal).

Efect: Defineşte o unitate de program ca subprogram extern având numele indicat prin *nume* şi utilizând parametri formali *p*. Transferarea valorilor se efectuează prin corespondența parametrilor formali *p* (ca ordine şi tip) cu parametrii efectivi din linia de apel. A se vedea instrucțiunea `CALL`.

TYPE, scriere secvențială:

```
TYPE f[,listă]
```

unde: `TYPE` – instrucțiunea executabilă;
`f` – referință la format (specificator de format);
`listă` – listă de ieşire.

Efect: Scrie o înregistrare pe dispozitivul de ieşire implicit (consolă/monitor), conținând valorile elementelor din *listă*. Valorile sunt convertite în conformitate cu specificația de format *f*.

Varianta cu format implicit: `TYPE *[,listă]`

UNLOCK, deblocarea unității logice:

```
UNLOCK([UNIT=]u[,ERR=e])
```
sau
```
UNLOCK u
```

unde:
- `UNLOCK` – instrucțiunea executabilă;
- `UNIT` – cuvânt cheie pentru desemnarea unității logice;
- `u` – expresie întreagă (semnificând numărul unei unități logice);
- `ERR` – cuvânt cheie pentru tratarea unei erori la executarea instrucțiunii;
- `e` – eticheta unei instrucțiuni executabile.

Efect: Deblochează toate înregistrările curente blocate, de pe unitatea logică *u*. Atunci când în sintaxă s-a folosit `ERR=`*e* şi apare o eroare la execuție, controlul va fi transferat instrucțiunii executabile cu eticheta *e* din cadrul aceleiaşi unități de program.

VIRTUAL, declararea memoriei virtuale:

```
VIRTUAL a(d)[,a(d)]...
```

unde:
- `VIRTUAL` – instrucțiunea declarativă;
- `a` – numele tabloului;
- `d` – declaratorul de dimensiune, sub forma:

  `n[,n]...`

  unde: `n` – este o expresie întreagă (semnificând numărul de elemente în dimensiunea respectivă).

Efect: Specifică rezervarea spațiului necesar memorării tablourilor indicate prin *a*(*d*), în afara spațiului direct adresabil al programului.

WRITE, scriere secvențială:

```
WRITE([UNIT=]u[,[FMT=]f][,ERR=e]) [listă]
```

unde:
- `WRITE` – instrucțiunea executabilă;
- `UNIT` – cuvânt cheie pentru desemnarea unității logice;
- `u` – expresie întreagă (semnificând numărul unei unități logice);
- `FMT` – cuvânt cheie pentru referința de format;
- `f` – referință la format (specificator de format);
- `ERR` – cuvânt cheie pentru tratarea unei erori la executarea instrucțiunii;
- `e` – eticheta unei instrucțiuni executabile;

`listă` – listă de ieşire.

Efect: Scrie una sau mai multe înregistrări pe unitatea logică *u*, conţinând valorile elementelor din *listă*. Valorile sunt convertite conform referinţei de format *f* (dacă aceasta a fost specificată). Atunci când în sintaxă s-a folosit `ERR=`*e* şi apare o eroare la execuţie, controlul va fi transferat instrucţiunii executabile cu eticheta *e* din cadrul aceleiaşi unităţi de program.

Varianta cu format implicit:

```
WRITE([UNIT=]u,[FMT=]*[,ERR=e]) [listă]
```

Efect: Scrie una sau mai multe înregistrări pe unitatea logică *u*, conţinând valorile elementelor din *listă*. Valorile sunt convertite în conformitate cu tipul elementelor din *listă*.

Varianta fără format:

```
WRITE([UNIT=]u[,ERR=e]) [listă]
```

Efect: Scrie una sau mai multe înregistrări pe unitatea logică *u*, conţinând valorile elementelor din *listă*.

WRITE, scriere în acces direct:

```
WRITE([UNIT=]u,REC=r[,[FMT=]f][,ERR=e]) [listă]
```

sau

```
WRITE(u'r[,[FMT=]f][,ERR=e]) [listă]
```

unde: `WRITE` – instrucţiunea executabilă;
`UNIT` – cuvânt cheie pentru desemnarea unităţii logice;
`u` – expresie întreagă (semnificând numărul unei unităţi logice);
`REC` – cuvânt cheie pentru desemnarea înregistrării;
`r` – expresie întreagă (semnificând numărul înregistrării vizate);
`FMT` – cuvânt cheie pentru referinţa de format;
`f` – referinţă la format (specificator de format);
`ERR` – cuvânt cheie pentru tratarea unei erori la executarea instrucţiunii;
`e` – eticheta unei instrucţiuni executabile;
`listă` – listă de ieşire (cu elemente separate prin virgulă).

Efect: Scrie una sau mai multe înregistrări pe unitatea logică *u*, conţinând valorile elementelor din *listă*, începând de la înregistrarea *r*. Valorile sunt convertite în conformitate cu referinţa de format *f* (dacă aceasta s-a

specificat în cadrul instrucțiunii). Atunci când în sintaxă s-a folosit ERR=*e* şi apare o eroare la execuție, controlul va fi transferat instrucțiunii executabile cu eticheta *e* din cadrul aceleiaşi unități de program.

Varianta fără format:

```
WRITE([UNIT=]u,REC=r[,ERR=e]) [listă]
```
sau
```
WRITE(u'r[,ERR=e]) [listă]
```

Efect: Scrie înregistrarea *r* pe unitatea logică *u* conținând valorile elementelor din *listă*.

Notă: Scrierea directă se aplică fişierelor cu organizare relativă deschise în acces direct. A se vedea instrucțiunea OPEN.

WRITE, scriere indexată:

```
WRITE([UNIT=]u[,[FMT=]f][,ERR=e]) [listă]
```

unde: WRITE – instrucțiunea executabilă;
UNIT – cuvânt cheie pentru desemnarea unității logice;
u – expresie întreagă (semnificând numărul unei unități logice);
FMT – cuvânt cheie pentru referința de format;
f – referință la format (specificator de format);
ERR – cuvânt cheie pentru tratarea unei erori la executarea instrucțiunii;
e – eticheta unei instrucțiuni executabile;
listă – listă de ieşire.

Efect: Scrie una sau mai multe înregistrări pe unitatea logică *u* (conectată la un fişier indexat), conținând valorile elementelor din *listă*. Valorile sunt convertite conform referinței de format *f* (dacă aceasta a fost specificată). Atunci când în sintaxă s-a folosit ERR=*e* şi apare o eroare la execuție, controlul va fi transferat instrucțiunii executabile cu eticheta *e* din cadrul aceleiaşi unități de program.

Varianta fără format:

```
WRITE([UNIT=]u[,ERR=e]) [listă]
```

Efect: Scrie una sau mai multe înregistrări pe unitatea logică *u* (conectată la un fişier indexat), conținând valorile elementelor din *listă*.

Notă: Scrierea indexată corespunde sintactic cu scrierea secvențială.

WRITE, scriere internă:

```
WRITE([UNIT=]c,[FMT=]f[,ERR=e]) [listă]
```

unde: `WRITE` – instrucţiunea executabilă;
`UNIT` – cuvânt cheie pentru desemnarea unităţii logice;
`c` – specificator de fişier intern;
`FMT` – cuvânt cheie pentru referinţa de format;
`f` – referinţă la format (specificator de format);
`ERR` – cuvânt cheie pentru tratarea unei erori la executarea instrucţiunii;
`e` – eticheta unei instrucţiuni executabile;
`listă` – listă de ieşire (elementele separate prin virgulă).

Efect: Scrie elementele din *listă* pe fişierul intern specificat prin *c*, convertindu-le în şiruri de caractere în conformitate cu specificaţia de format *f*. Atunci când în sintaxă s-a folosit `ERR=`*e* şi apare o eroare la execuţie, controlul va fi transferat instrucţiunii executabile cu eticheta *e* din cadrul aceleiaşi unităţi de program.

Notă: Scrierea internă se poate utiliza pentru convertirea valorilor de tip întreg în caractere (chiar în caractere extinse, neimprimabile), cu condiţia ca valorile să corespundă cu codul ASCII. A se vedea şi instrucţiunea `ENCODE`.

3.4 DESCRIPTORII DE INTRARE/IEŞIRE

Descriptorii sunt specificatori de câmpuri utilizaţi la operaţiile de citire/scriere în cadrul instrucţiunii `FORMAT` pentru stabilirea formei valorilor din lista de intrare/ieşire. Ei pot fi folosiţi şi fără instrucţiunea `FORMAT`, fiind citaţi în asemenea cazuri în lista de parametri a instrucţiunilor de intrare ieşire, în locul referinţei de format *f*, sub forma: `'(listă)'` (unde *listă* are semnificaţia prezentată la instrucţiunea `FORMAT`). Lista descriptorilor trebuie să corespundă ca ordine şi tip cu elementele din lista de intrare/ieşire, în caz contrar putând avea loc conversii, efecte nedorite sau erori.

Descriptorii pot alcătui liste complexe, repetiţiile putând fi marcate prin includerea în paranteze rotunde precedate de un factor de repetiţie opţional (scalar). Dacă numărul descriptorilor specificaţi în listă este mai mic decât numărul valorilor din lista de intrare/ieşire, se vor relua de la stânga la dreapta descriptorii din ultima paranteză deschisă.

În cele ce urmează, vom prezenta sub formă tabelară descriptorii din Fortran 77 în ordine alfabetică. Aceştia se regăsesc şi în variantele ulterioare ale limbajului pe lângă alţi descriptori mai noi bineînţeles, ei putând fi grupaţi în două categorii: *de editare* şi *de control*.

Tabel cu descriptorii de editare:

Tip	Sintaxă	Efect
A	`[n]Aw`	Descrie *n* câmpuri de tip alfanumeric (`CHARACTER`), fiecare cu *w* poziții.
D	`[n]Dw.d`	Descrie *n* câmpuri de numere reale extinse (`DOUBLE PRECISION` sau `REAL*8`), fiecare cu câte *w* poziții din care *d* sunt după punctul zecimal.
E	`[n]Ew.d[Ee]`	Descrie *n* câmpuri de numere reale sub formă exponențială, fiecare cu câte *w* poziții din care *d* sunt după punctul zecimal. Opțional se poate specifica prin *e* numărul de caractere utilizate pentru exponent.
F	`[n]Fw.d`	Descrie *n* câmpuri de numere reale, fiecare cu câte *w* poziții din care *d* sunt după punctul zecimal.
G	`[n]Gw.d[Ee]`	Descrie *n* câmpuri de numere reale extinse (`DOUBLE PRECISION` sau `REAL*8`) sub formă exponențială, fiecare cu câte *w* poziții din care *d* sunt după punctul zecimal. Opțional se poate specifica prin *e* numărul de caractere utilizate pentru exponent.
H	`cHa[a...]`	Descrie o constantă de tip *Hollerith* având lungimea *c* şi conținutul specificat prin *a* (un număr de *c* caractere ce urmează după litera cheie `H`).
I	`[n]Iw`	Descrie *n* câmpuri de numere întregi, fiecare cu câte *w* poziții.
L	`[n]Lw`	Descrie *n* câmpuri de tip logic, fiecare cu câte *w* poziții.
Z	`[n]Zw`	Descrie *n* câmpuri de numere hexadecimale, fiecare cu câte *w* poziții.

Tabel cu descriptorii de control:

Tip	Sintaxă	Efect
/	`/[/...]`	Similar cu returul de car (*Enter*). Determină încheierea rândului curent şi saltul la începutul unui rând nou pentru tratarea următoarelor valori prin descriptorii din lista curentă.
\	`\[\...]`	Inhibă saltul la rând nou. Următoarele valori formatate vor fi tot pe rândul curent, în continuare.
BN	`BN`	Ignoră spațiile dintr-un câmp numeric (*Blank None*).
BZ	`BZ`	Determină interpretarea spațiilor dintr-un câmp numeric ca zerouri (*Blank Zero*).
S	`S`	Determină tratarea semnului valorilor numerice (*Sign*), anulând efectul descriptorilor `SP` şi `SS`.
SP	`SP`	Determină afişarea semnului plus (`+`) la valori numerice pozitive (*Sign Plus*).
SS	`SS`	Inhibă semnul opțional plus (`+`) la valori numerice pozitive (*Sign Suppressed*).

Tip	Sintaxă	Efect
T	`Tn`	Introduce *n* tabulatori orizontali (*Tab*).
TL	`TLn`	Introduce *n* tabulatori orizontali spre stânga (*Tab Left*).
TR	`TRn`	Introduce *n* tabulatori orizontali spre dreapta (*Tab Right*).
X	`nX`	Sare peste *n* poziții (introduce *n* spații)

Notă: Descriptorii de formatare `/` şi `\` nu trebuie separați neapărat prin virgulă de restul descriptorilor din listă, ei înşişi având şi rol de separare.

3.5 FUNCȚIILE INTRINSECI DIN FORTRAN 77

Funcțiile intrinseci sunt specifice bibliotecilor utilizate, având nume simbolice prestabilite (rezervate). Printre ele există unele ce fac parte din echiparea standard a mediului de programare, regăsindu-se în toate variantele limbajului Fortran. Numele acestor funcții sunt rezervate, nu pot exista variabile sau tablouri de variabile care să aibe nume ce coincid cu cele ale funcțiilor intrinseci. De asemenea, numele acestor funcții nu se recomandă să apară într-o listă a unei instrucțiuni `EXTERNAL`, acest fapt ducând la anularea definiției lor intrinseci. În cazul includerii numelor lor în liste ale instrucțiunii declarative `INTRINSIC`, ele vor putea fi utilizate ca parametri la proceduri (unități de subprograme sau de funcții definite de utilizator).

Sintaxa generală a acestora este următoarea:

```
nume(p[,p]...)
```

unde: `nume` – nume simbolic (denumirea funcției);
`p` – nume simbolic (al parametrului efectiv).

Tabel cu funcțiile intrinseci din limbajul Fortran 77:

Definiție	Nume generic	Parametrii		Funcția		Efect
		Nr.	Tip	Nume	Tip	
|x|	`ABS`	1	R	`ABS`	R	Returnează valoarea absolută (modulul) argumentului specificat.
			D	`DABS`	D	
			R	`CABS`	R	
			I	`IIABS`	I	
			I4	`JIABS`	I4	
	`IABS`		I	`IIABS`	I	
			I4	`JIABS`	I4	
cos(x)	`COS`	1	R	`COS`	R	Returnează valoarea cosinusului argumentului exprimat în radiani.
			D	`DCOS`	D	
			C	`CCOS`	C	
sin(x)	`SIN`	1	R	`SIN`	R	Returnează valoarea sinusului argumentului exprimat în radiani.
			D	`DSIN`	D	
			C	`CSIN`	C	

Definiție	Nume generic	Parametrii		Funcția		Efect
		Nr.	Tip	Nume	Tip	
e^x	EXP	1	R	EXP	R	Returnează valoarea exponențială a argumentului.
			D	DEXP	D	
			C	CEXP	C	
ln(x)	LOG	1	R	ALOG	R	Returnează valoarea logaritmului natural al argumentului.
			D	DLOG	D	
			C	CLOG	C	
log(x)	LOG10	1	R	ALOG10	R	Logaritmul în baza 10 al argumentului.
			D	DLOG10	D	
$\sqrt{x}$	SQRT	1	R	SQRT	R	Returnează radicalul argumentului.
			D	DSQRT	D	
			C	CSQRT	C	
tg(x)	TAN	1	R	TAN	R	Tangenta argumentului exprimat în radiani.
			D	DTAN	D	
arccos(x)	ACOS	1	R	ACOS	R	Arccosinusul argumentului exprimat în radiani.
			D	DACOS	D	
arcsin(x)	ASIN	1	R	ASIN	R	Arcsinusul argumentului exprimat în radiani.
			D	DASIN	D	
arctg(x)	ATAN	1	R	ATAN	R	Arctangenta argumentului exprimat în radiani.
			D	DATAN	D	
arctg(x/y)	ATAN2	2	R	ATAN2	R	Arctangenta argumentului exprimat în radiani.
			D	DATAN2	D	
cosh(x)	COSH	1	R	COSH	R	Cosinusul hiperbolic al argumentului.
			D	DCOSH	D	
sinh(x)	SINH	1	R	SINH	R	Sinusul hiperbolic al argumentului.
			D	DSINH	D	
tgh(x)	TANH	1	R	TANH	R	Tangenta hiperbolică a argumentului.
			D	DTANH	D	
rest din x/y	MOD	2	I	IMOD	I	Returnează restul împărțirii dintre argumente (cu semnul primului argument).
			I4	JMOD	I4	
			R	AMOD	R	
			D	DMOD	D	
max(x,y,...)	MAX	>1	I	IMAX0	I	Returnează valoarea maximă dintre elementele cuprinse în lista de argumente.
			I4	JMAX0	I4	
			R	AMAX1	R	
			D	DMAX1	D	
	MAX0		I	IMAX0	I	Valoarea maximă dintr-o listă de valori întregi.
			I4	JMAX0	I4	
	MAX1		R	IMAX1	I4	Maximul dintr-o listă de valori reale, ca întreg.
			R	JMAX1	I4	
	AMAX0		I	AIMAX0	R	Maximul dintr-o listă de valori întregi, ca real.
			I4	AJMAX0	R	

Definiție	Nume generic	Parametrii		Funcția		Efect
		Nr.	Tip	Nume	Tip	
min(x,y,...)	MIN	>1	I	IMIN0	I	Returnează valoarea minimă dintre elementele cuprinse în lista de argumente.
			I4	JMIN0	I4	
			R	AMIN1	R	
			D	DMIN1	D	
	MIN0		I	IMIN0	I	Valoarea minimă dintr-o listă de valori întregi.
			I4	JMIN0	I4	
	MIN1		R	IMIN1	I4	Minimul dintr-o listă de valori reale, ca întreg.
			R	JMIN1	I4	
	AMIN0		I	AIMIN0	R	Minimul dintr-o listă de valori întregi, ca real.
			I4	AJMIN0	R	
trunchiere [x]	INT	1	R	IINT	I	Returnează valoarea trunchiată a argumentului la cel mai apropiat întreg.
			R	JINT	I4	
			D	IIDINT	I	
			D	JIDINT	I4	
	IDINT		D	IIDINT	I	
			D	JIDINT	I4	
	AINT		R	AINT	R	Valoarea reală trunchiată (cu zero la zecimale).
			D	DINT	D	
rotunjire la cel mai apropiat întreg [x + 0.5*sign(x)]	NINT	1	R	ININT	I	Returnează valoarea rotunjită a argumentului la cel mai apropiat întreg.
			R	JNINT	I4	
			D	IIDNNT	I	
			D	JIDNNT	I4	
	IDNINT		D	IIDNNT	I	
			D	JIDNNT	I4	
	ANINT		R	ANINT	R	Valoarea reală rotunjită (cu zero la zecimale).
			D	DNINT	D	
x – min(x,y)	DIM	2	I	IIDIM	I	Returnează valoarea diferenței dintre cele două argumente, dacă aceasta este pozitivă. Altfel returnează zero.
			I4	JIDIM	I4	
			R	DIM	R	
			D	DDIM	D	
	IDIM		I	IIDIM	I	
			I4	JIDIM	I4	
transferul semnului între două valori	SIGN	2	I	IISIGN	I	Returnează valoarea primului argument cu semnul celuilalt: SIGN(y)*ABS(x)
			I4	JISIGN	I4	
			R	SIGN	R	
			D	DSIGN	D	
	ISIGN		I	IISIGN	I	
			I4	JISIGN	I4	
generarea unui număr	RAN	1	I4	RAN	R	Returnează un număr pseudoaleator cu distribuție uniformă între 0 și 1.
		2	I	RAN	R	

Definiție	Nume generic	Parametrii		Funcția		Efect
		Nr.	Tip	Nume	Tip	
conversii de valori, între diferite tipuri	FLOAT	1	I	FLOATI	R	Conversie întreg în real.
			I4	FLOATJ	R	
	DFLOAT	1	I	DFLOTI	D	Conversie întreg în dublă precizie.
			I4	DFLOTJ	D	
	IFIX	1	R	IIFIX	I	Conversie real în întreg prin rotunjire.
			R	JIFIX	I4	
	SNGL	1	D	SNGL	R	Conversie în real (simplă precizie).
			I	FLOATI	R	
			I4	FLOATJ	R	
	DBLE	1	R	DBLE	D	Conversie în dublă precizie.
			I	DFLOTI	D	
			I4	DFLOTJ	D	
	CMPLX	1,2	I	–	C	Conversie în valoare complexă a argumentului (sau în parte reală şi imaginară a argumentelor).
			I4	–	C	
			R	CMPLX	C	
			D	–	C	
	ICHAR	1	CH	ICHAR	I	Conversie în codul ASCII.
parte reală	REAL	1	C	REAL	R	Returnează partea reală a argumentului.
			I	FLOATI	R	
			I4	FLOATJ	R	
			D	SNGL	R	
parte imaginară	AIMAG	1	C	AIMAG	R	Returnează partea imaginară dintr-un număr complex.
conjugare	CONJG	1	C	CONJG	C	Conjugatul unui complex.
produs, lungime dublă	DPROD	2	R	DPROD	D	În dublă precizie produsul celor două argumente reale.
ŞI logic pe bit	IAND	2	I	IIAND	I	AND logic între două argumente.
			I4	JIAND	I4	
SAU logic pe bit	IOR	2	I	IIOR	I	OR logic inclusiv între două argumente.
			I4	JIOR	I4	
SAU exclusiv pe bit	IEOR	2	I	IIEOR	I	OR logic exclusiv între două argumente.
			I4	JIEOR	I4	
negația logică pe bit	NOT	1	I	INOT	I	Complementul logic al argumentului.
			I4	JNOT	I4	
deplasare logică pe bit	ISHFT	2	I	IISHFT	I	Deplasarea terminală logică a biților din argument.
			I4	JISHFT	I4	
lungimea unui şir	LEN	1	CH	LEN	I	Numărul de caractere din şirul considerat argument.
poziția într-un şir a unui subşir	INDEX	2	CH	INDEX	I	Poziția de început a subşirului în şirul specificat ca primul argument.

Definiție	Nume generic	Parametrii		Funcția		Efect
		Nr.	Tip	Nume	Tip	
comparație lexicală	–	2	CH	LLT	L	Returnează o valoare logică rezultată dintr-o comparație între argumente de tip caracter.
			CH	LLE	L	
			CH	LGT	L	
			CH	LGE	L	

Notă: Pentru tipul parametrilor şi a funcțiilor s-au utilizat următoarele notații:

I – INTEGER*2
I4 – INTEGER*4
R – REAL*4
D – DOUBLE PRECISION (REAL*8)
CH – CHARACTER
C – COMPLEX
L – LOGICAL*2

Observații:

- Argumentul funcției logaritmice reale sau dublă precizie trebuie să fie pozitiv, iar argumentul funcției CLOG trebuie să fie diferit de (0.,0.).
- Argumentul funcției rădăcină pătrată reală sau dublă precizie trebuie să fie pozitiv sau nul. Valoarea funcției CSQRT are întotdeauna partea reală mai mare sau egală cu zero (când partea reală a valorii funcției este zero, atunci partea sa imaginară este mai mare sau egală cu zero).
- Rezultatul funcțiilor ATAN şi DATAN este în intervalul –π/2...π/2, iar ale funcțiilor ATAN2 şi DATAN2 în intervalul –π...π, semnul fiind dat de primul argument. Dacă ambele argumente sunt nule atunci rezultatul este nedefinit.
- Funcția MOD(x,y) fiind definită prin expresia x-|x/y|*y, rezultă nedefinită dacă al doilea argument este nul.
- Funcția de transfer al semnului este nedefinită dacă al doilea argument este nul.
- Dacă funcția CMPLX are un singur argument, acesta este convertit şi atribuit părții reale a valorii complexe rezultate, partea imaginară rezultând nulă.

CAPITOLUL 4: FORTRAN 90

4.1 TRECEREA DE LA FORTRAN 77 LA FORTRAN 90

În prezent se lucrează la un compilator GNU Fortran 90, ceea ce impune discutarea unor aspecte legate de scrierea programelor într-o manieră mai modernă. Dacă se compară posibilitățile oferite de Fortran 77 cu cele din alte limbaje de programare (Forth, C, C++, Pascal etc.) se observă foarte uşor motivul pierderii interesului programatorilor pentru acest limbaj. Dezvoltarea acestuia a survenit în mod firesc, ca o necesitate pentru susținerea şi adaptarea programelor deja existente pe lângă realizarea de aplicații noi, moderne şi competitive, în pas cu dezvoltarea domeniillor ştiințifice conexe şi cu cerințele utilizatorilor de programe. Aspectele imputate în general versiunii consacrate a limbajului Fortran 77 sunt următoarele:

- Lipsa facilităților de stocare dinamică;
- Lipsa tipurilor şi structurilor de date definite de utilizator (exceptând blocul `COMMON`, considerat chiar revoluționar la momentul apariției);
- Uşurința cu care se pot comite greşeli nesesizate de compilator, în special la apelarea procedurilor (subprograme sau funcții);
- Portabilitatea uneori limitată a programelor (de multe ori conținând caracteristici dependente de platformă, mai ales în cazul folosirii extensiilor specifice);
- Structurile de control sunt slabe (de multe ori nu se pot evita instrucțiunile de salt ceea ce conduce la complicarea înțelegerii codului);
- Reguli arhaice rămase din era cartelelor perforate (format fix pe 80 de coloane, nume simbolice limitate la 6 caractere, etc.).

Schimbările esențiale aduse de limbajul Fortran 90 sunt considerate a fi următoarele:

- Format liber la scrierea sursei, concomitent cu alte îmbunătățiri simple;
- Tablouri ca şi obiecte, expresii de tablouri, asignări şi funcții;
- Alocare dinamică a memoriei, pointeri ce permit construcții de structuri complexe şi dinamice de date;
- Tipuri de date definite de utilizator (pe lângă posibilitatea definirii unor operatori noi, operatorii existenți pot fi redefiniți prin *supraîncărcare*);
- Modulul – o unitate de program care poate conține date şi seturi de proceduri conexe (subprograme sau funcții). Se pot implementa clase şi funcții aparținătoare pentru programarea orientată pe obiecte;
- Procedurile pot fi recursive, pot avea nume generice, argumente opționale etc.;
- Structuri noi de control (cum ar fi: `SELECT CASE`, `CYCLE`, `EXIT`) care permit restrângerea utilizării etichetelor şi a salturilor explicite.

Programele pot fi redactate astfel într-o manieră mai simplă, uşurând întreținerea lor. Codul generat poate deveni mai sigur şi mai stabil deoarece compilatorul poate detecta mult mai multe greşeli în cazul utilizării caracteristicilor de securitate avansată. Programele sunt mai portabile, rămânând foarte puține trăsături legate de maşină, nevoia de a utiliza extensii specifice unor platforme fiind redusă. Se pot scrie chiar aplicații pentru procesare paralelă,

asemenea operații fiind suportate pe lângă celelalte trăsături noi. În aceeaşi timp Fortran 77 rămâne un subset acceptat pe deplin, deci noile trăsături se pot adopta gradual în funcție de nevoile ivite. Există însă anumite extensii obişnuite ale limbajului Fortran 77 care nu au fost incluse în varianta 90 de bază, dintre care menționăm:

- Formatul tabular (caracterul de tabulare <TAB> se converteşte în spații);
- Declarațiile de tipul `INTEGER*2` şi `REAL*8` (sintaxa nouă este mai bună dar mai complicată);
- Constante hexadecimale, octale şi binare în expresii (se admit doar în declarațiile de tip `DATA`);
- Structuri de date VAX (sintaxa structurilor este diferită în Fortran 90);
- Expresii în cadrul instrucțiunilor `FORMAT` (se pot realiza indirect cu operații interne de intrare/ieşire);
- Anumite opțiuni de la instrucțiunea `OPEN` (de exemplu `ACCESS='APPEND'` schimbat cu `POSITION='APPEND'`).

Dacă se foloseşte varianta curată a standardului Fortran 77, atunci nu apar probleme la compilarea după standardul Fortran 90. Deşi în Fortran 90 nu există cuvinte strict rezervate, sunt disponibile 75 de funcții intrinseci noi față de versiunea anterioară a limbajului. Problemele legate de eventuale coincidențe la numele funcțiilor se pot evita folosind instrucțiunea declarativă `EXTERNAL`.

4.1.1 Compilatoare

Compilatoarele Fortran 77 realizează în general stocarea variabilelor în mod static, deci omiterea opțiunii `SAVE` nu avea repercursiuni semnificative. Majoritatea sistemelor Fortran 90 stochează însă local variabilele, în proceduri (prin stivă), folosind alocare statică doar atunci când e nevoie (cum ar fi cazul variabilelor cu o valoare inițială, sau cu atributul `SAVE` explicit), aşa că omiterea opțiunii `SAVE` în cadrul surselor vechi poate crea anumite probleme.

Deşi există deja o varietate mare de compilatoare Fortran 90, atât comerciale (ce-i drept mai scumpe decât compilatoarele Fortran 77), cât şi cu licență liberă, necomerciale, nu toate sunt eficiente sau stabile. Dintre cele accesibile la acest moment menționăm compilatorul F (fiind practic realizat pe baza unui subset din Fortran 90, produs de firma Imagine1, disponibil la ora actuală sub licență liberă doar pentru sistemul de operare Linux, disponibil la http://www.imagine1.com/imagine1/). Din păcate, compilatorul ELF90 (o variantă mai restrânsă a compilatorului Fortran 90, creată de firma Lahey, http://www.lahey.com/), la ora actuală nu mai este promovat. Deşi se lucrează la compilatorul GNU Fortran 90, va mai trece ceva timp până când acesta va fi finalizat. Predecesorul său, compilatorul GNU Fortran 77 (g77) este poate cel mai cunoscut deocamdată, fiind disponibil pentru mai multe platforme de lucru. Este stabil şi accesibil dar, deşi suportă complet standardul Fortran 77, nu acceptă decât câteva dintre trăsăturile noi introduse prin limbajul Fortran 90. Există şi traducătoare între diferite versiuni ale limbajului Fortran (cum ar fi cel creat de compania Pacific-Sierra Research, http://www.psrv.com/, pentru a traduce codul sursă din Fortran 90 în Fortran 77, în scopul compilării cu g77).

4.1.2 Diferențe formale

Sursa unui program poate fi scrisă în Fortran 90 atât în formă liberă, cât şi în formă fixă, însă cele două formate nu pot fi amestecate. În general compilatoarele consideră fişierele sursă cu extensia `.F90` ca fiind implicit în formă liberă. Liniile din cadrul sursei pot avea o lungime maximă de 132 de caractere (incluzând şi spațiile). În cazul unor rânduri mai lungi întreruperea se va marca prin caracterul ampersand (**&**) scris la capătul rândului incomplet. Pe rândul incomplet, după marcajul de continuare se pot scrie comentarii (cu condiția să nu se depăşească cele 132 de caractere în total). Dacă prin întreruperea rândului s-a despărțit un nume simbolic sau o constantă, atunci următoarea linie (rândul de continuare) trebuie să înceapă cu caracterul **&**. Cei ce doresc să scrie cod valabil în ambele formate (se poate dovedi util folosind instrucțiunea INCLUDE atât pentru surse vechi cât şi pentru surse de tipul Fortran 90), pot marca continuarea unui rând după coloana 72 (se va considera comentariu în formatul vechi) în acelaşi timp scriind caracterul **&** şi în coloana 6 din rândul de continuare.

În formatul liber spațiile (caracterele blank) sunt semnificative, ele nu pot apărea în nume simbolice sau constante (cu excepția valorii citate a constantelor de tip caracter), având rol de separare în anumite cazuri. La scrierea codului sursă se pot folosi atât litere mici cât şi litere mari (mărimea caracterelor contează doar în cadrul valorii constantelor de tip caracter). Numele simbolice pot avea lungimi de până la 31 de caractere, putând conține pe lângă caracterele alfanumerice şi caracterul de subliniere (_). Instrucțiunile scrise pe un rând comun trebuie să fie separate prin caracterul punct-virgulă (`;`). Comentariile se marchează cu semnul exclamării în față, ele putând începe oriunde în cadrul unui rând (chiar şi după o instrucțiune, dar în cazul formatului fix marcajul nu poate fi în coloana 6). Constantele de tip caracter pot fi citate ca şi conținut prin delimitarea lor fie cu caracterul apostrof (`'`) fie folosind caracterul ghilimele (`"`). Operatorii relaționali pot fi scrişi atât sub forma veche, cât şi sub forma nouă (a se vedea tabelul corespunzător din capitolul precedent).

4.1.3 Specificări noi

Declarația IMPLICIT NONE este comună în Fortran 90, fiind recomandată şi pentru verificarea mai temeinică la compilare. Declarația de tipul DOUBLE PRECISION este doar un caz special al declarației de tip REAL, în consecință şi datele de tip complex pot fi reprezentate în mod firesc pe lungime dublă (DOUBLE COMPLEX). Deşi se sprijină în continuare instrucțiunea INCLUDE, o varianta nouă: MODULE, oferă mai multe facilități. Declarațiile de tip acceptă o sintaxă nouă (cu separator alcătuit din două puncte duble între declarații şi listă), permițând definirea simultană a tuturor atributelor pentru elementele unei liste, precum şi inițializarea simultană, de exemplu:

```
INTEGER, DIMENSION(10,10) :: x, y, z
REAL, PARAMETER :: pi = 3.12159, ud = 180.0/pi
CHARACTER(LEN=12) :: fis = "intrare1.dat"
```

Declarația DATA devine astfel aproape redundantă (rămâne utilă la inițializarea unor zone parțiale de tablou, a unor constante hexadecimale etc), iar atributul SAVE este aplicat

implicit asupra tuturor variabilelor cu valori inițiale (indiferent dacă ele sunt declarate prin tip sau DATA). Evoluția sintaxei permite alcătuirea unor adevărate expresii de specificare. Atributul INTENT permite declararea intenției de folosire a unor argumente formale (valorile admise fiind: IN, OUT şi INOUT).

Declarațiile de tip `LOGICAL*1`, `INTEGER*2`, sau `REAL*8` erau extensii obişnuite ale limbajului Fortran 77, dar ele nu se regăsesc în Fortran 90. Această variantă mai nouă a limbajului dispune de cinci tipuri intrinseci de valori (tip caracter, logic, întreg, real şi complex) admițând diferite feluri (specificate explicit prin atributul KIND) ale acestor tipuri. Pentru valorile de tip complex şi real există două feluri predefinite (al doilea corespunzând reprezentării pe lungime dublă). Felul se poate specifica printr-o valoare întreagă, corespunzătoare teoretic lungimii de reprezentare (standardul limbajului nu precizează semnificația acestei valori întregi), cum ar fi de exemplu `INTEGER(2)` în loc de `INTEGER*2`. Pentru a oferi o flexibilitate şi implicit o portabilitate mărită programelor, există două funcții intrinseci noi: SELECTED_INT_KIND (care selectează o valoare întreagă pentru numărul minim de cifre zecimale dorite) şi SELECTED_REAL_KIND (care selectează pentru valori reale precizate numărul cifrelor zecimale şi domeniul pentru exponent). Astfel:

```
INTEGER, PARAMETER :: &
scurt = SELECTED_INT_KIND(4), &      ! intregi cu >= 4 cifre
lung = SELECTED_INT_KIND(9), &       ! intregi cu >= 9 cifre
dublu = SELECTED_REAL_KIND(15, 200)  ! valori reale cu 15 cifre si
                                     ! cu domeniul 10**200
INTEGER(scurt) :: poza(1024,768)
INTEGER(lung) :: numar
REAL(dublu) :: tablou(20,20)
```

Cea mai bună metodă este de a include definițiile parametrilor de fel (ca cele de mai sus) într-un *modul* care să fie utilizat de-a lungul programului. La rândul lor, şi constantele pot avea ataşate atribute de fel, acolo unde corespondența felului este cerută (de exemplu în cadrul parametrilor unei proceduri). Iată un exemplu cu apelarea unui subprogram, folosind pe post de parametri efectivi valori cu specificație de fel (acestea fiind despărțite de constante prin caracterul _):

```
CALL subprogram( 3.14159265358_dublu, 12345_lung, 42_scurt)
```

Menționăm că felul parametrilor (*dublu*, *lung*, *scurt*) din acest apel este cel definit în exemplul anterior.

Funcția `KIND` returnează la rândul său parametrul de fel al oricărei variabile, de exemplu:

```
WRITE(*,*) " felul pentru dubla precizie este ", KIND(0d0)
```

În principiu, regula de fel poate fi extinsă şi la valorile de tip caracter, sistemele Fortran putând suporta seturi de caractere pe 16 biți (cum este setul Unicode).

4.2 STRUCTURI DE CONTROL NOI, INTRODUSE ÎN FORTRAN 90

În varianta anterioară a limbajului Fortran nu exista instrucțiune corespunzătoare reprezentării primitivei alternative generalizate. Pentru traducerea unei asemenea structuri din pseudocod sau dintr-o schemă logică era nevoie fie de un set de instrucțiuni de salt calculate cu multitudinea de etichete corespunzătoare, fie de un set de instrucțiuni IF (ELSE IF) care rezulta destul de complexă. În Fortran 90, asemenea structuri se pot scrie foarte uşor prin utilizarea instrucțiunii SELECT CASE. Iată un exemplu preluat din literatura de specialitate de limbă engleză, referitor la alegerea indicelui corespunzător unui număr de ordine pentru zilele unei luni:

```
SELECT CASE(numar_zi)
 CASE(1, 21, 31)             ! cazul numerelor ce se termină cu unu
       indice = ’st’
 CASE(2, 22)                 ! cazul numerelor terminate cu doi
       indice = ’nd’
 CASE(3, 23)                 ! cazul numerelor terminate cu trei
       indice = ’rd’
 CASE(4:20, 24:30)           ! cazul celorlaltor numere
       indice = ’th’
 CASE DEFAULT                ! cazul implicit, cu valoare invalida
       indice = ’??’
       WRITE(*,*)’ data invalida: ’, numar_zi
END SELECT
WRITE(*, "(I4,A2)") numar_zi, indice
```

Expresiile utilizate pot fi de tip întreg sau caracter, însă domeniile precizate la instrucțiunile CASE nu pot să se suprapună sau să se intersecteze. Aceste domenii trebuie să fie foarte clar precizate. Tratarea cazului implicit (CASE DEFAULT) este opțională.

Şi în ceea ce priveşte instrucțiunile de ciclare/repetare s-au făcut îmbunătățiri semnificative. Instrucțiunea DO a fost actualizată în Fortran 90, eticheta ce marca sfârşitul modulului de repetat nemaifiind necesară, ea fiind înlocuită prin instrucțiunea END DO. În plus, instrucțiunea CYCLE va determina pornirea unui ciclu, iar instrucțiunea EXIT permite ieşirea din ciclu înainte de atingerea marcajului terminal (END CYCLE).

Există şi posibilitatea specificării unui ciclu DO nedefinit, adică fără un contor explicit, în acest caz ieşirea din ciclu trebuie asigurată prin EXIT. Se poate folosi şi instrucțiunea DO WHILE (corespunzătoare unei primitive repetitive pre-condiționate), aceasta va avea efect similar cu instrucțiunea DO nedefinită, după cum se poate observa şi în următorul exemplu:

```
DO WHILE( ABS(x - xmin) > 1.0e-5)    ! varianta cu ciclu DO-WHILE
CALL itereaza(x, xmin)
END DO
                                     !... sau
DO                                   ! aceeasi ciclu, cu DO nedefinit
IF( ABS(x - xmin) <= 1.0e-5) EXIT
CALL itereaza(x, xmin)
END DO
```

Pentru a facilita citirea şi înțelegerea programelor se pot boteza anumite structuri (cicluri DO, blocuri IF, structuri CASE) în cazul în care sunt prea complexe şi ar necesita EXIT (sau CYCLE) suplimentar. Aceste nume însă nu pot fi folosite ca etichete pentru instrucțiunile de salt GO TO. Iată un mic exemplu, cu numele *exterior* şi *interior* pentru două cicluri:

```
!...
suma = 0.0
exterior: DO j = 1,ny         ! suma pana la obtinerea lui zero
interior: DO i = 1,nx
IF(tablou(i,j) == 0.0) EXIT exterior
suma = suma + tablou(i,j)
END DO interior
END DO exterior
```

Etichetele marchează de regulă instrucțiunile la care se ajunge printr-un salt. Multitudinea salturilor poate complica claritatea programelor scrise, din acest motiv se recomandă utilizarea structurilor mai avansate acolo unde se poate. Folosirea etichetelor poate fi uşor evitată în următoarele situații:

- folosirea ciclurilor DO structurate (cu END DO), tratarea excepțiilor putându-se realiza prin instrucțiunile EXIT sau RETURN;
- înlocuirea instrucțiunilor de salt (GO TO) calculat prin structuri SELECT CASE;
- citarea listei de descriptori în locul specificării de format la instrucțiunile de citire sau scriere. De exemplu:

```
CHARACTER(LEN=2) :: raspuns
!...
WRITE(*,"(A)") "raspundeti cu DA sau NU :"
READ(*,"(A2)") raspuns
```

Funcțiile se puteau defini în Fortran 77 în două moduri: printr-o singură expresie (pe o linie) sau ca modul extern (subprogram FUNCTION). În Fortran 90 există însă posibilitatea declarării unor proceduri interne, într-o manieră structurată, ca în exemplul următor:

```
SUBROUTINE arie_poligon(laturi)      ! o procedura externa
IMPLICIT NONE                        ! valabil peste tot
!...
aria1 = arie_triunghi(a, b, x)
!...
aria2 = arie_triunghi(x, c, d)
!...
CONTAINS                             ! urmeaza procedura interna ...
REAL FUNCTION arie_triunghi(a, b, c) ! procedura interna
REAL, INTENT(IN) :: a, b, c
REAL :: s                            ! variabila locala in functie
s = 0.5 * (a + b + c)
arie_triunghi = sqrt(s * (s-a) * (s-b) * (s-c))
END FUNCTION arie_triunghi           ! sfarsitul procedurii interne
SUBROUTINE inclus                    ! o alta procedura interna
!...
END SUBROUTINE inclus                ! sfarsitul procedurii
END SUBROUTINE arie_poligon
```

Astfel, specificarea funcțiilor nu mai este restrânsă la doar o singură linie. Procedurile interne pot fi şi subprograme (nu numai funcții). La fel ca în exemplul precedent, ele trebuie să fie precedate de instrucțiunea declarativă CONTAINS care totodată marchează şi finalul descrierii procedurii externe gazdă. În cadrul unei unități de program se pot declara oricâte proceduri interne, care la rândul lor se pot apela şi reciproc, însă nu se admite combinarea (intersectarea) lor. Folosirea declarațiilor END SUBROUTINE şi END FUNCTION sunt obligatorii pentru a marca finalul specificațiilor, precizarea numelor procedurilor după aceste instrucțiuni fiind opțională (se recomandă doar pentru claritatea sursei).

Procedurile interne au acces la toate variabilele modulului gazdă (cu excepția cazurilor în care conțin declarații cu nume identice), dar acest aspect nu este valabil şi invers.

4.3 TABLOURI DE DATE

Tablourile au o semnificație şi un rol mai extins ca în varianta anterioară a limbajului, devenind obiecte de primă clasă asupra cărora se pot aplica operații prin expresii generice. Operațiile vor fi efectuate pentru fiecare element al tablourilor (fără a scrie cicluri explicite) cu condiția ca tablourile să fie conforme (formă şi dimensiuni compatibile). Dacă apar valori scalare într-o expresie, ele se vor aplica prin operatorii specificați, pe fiecare element al tablourilor vizate. De exemplu:

```
REAL :: scalar, vector(123), tablou(4,5,6)
CHARACTER(LEN=1), PARAMETER :: zi(0:6) = (/’D’,’L’,’M’,’M’,’J’,’V’,’S’/)
REAL, DIMENSION(768,1024) :: img, fundal, expunere, rezultat, std_err
!...
std_err = 0.0       ! toate elementele din acest tablou primesc valoarea
                    ! nulă
rezultat = (img - fundal) / expunere     ! expresie cu tablouri
fundal = 0.1 * expunere + 0.125   ! expresie cu tablouri şi scalari
!...
```

Noțiunea de zonă sau secțiune desemnează elemente de tablou referite prin indici de poziție sau un domeniu definit prin elementele de tablou componente. Declararea acestor zone se poate face prin numele tabloului de care aparțin şi ele pot fi inițializate cu valori cuprinse într-o listă (ca în exemplul de mai sus, la tabloul *zi*). Fiecare zonă sau element al tabloului moşteneşte proprietățile corespunzătoare de tip INTENT, PARAMETER şi TARGET ale tabloului, însă fără a moşteni atributul POINTER. La definirea secțiunilor nu trebuie să ne referim neapărat la zone contigue, fiind permise şi suprapuneri de elemente între diferite zone. De exemplu:

```
! img(2:101,301:500)  este o referinta la o zonă de 100x200 din tabloul img
! b(1:10:2)           este o referință la b(1), b(3), b(5), b(7) şi b(9)
                               !...
      a(2:10) = a(1:9)         ! translatare în sus cu un element
      b(1:9) = b(3:11)         ! translatare în jos cu doua elemente
```

Bineînțeles, în asemenea cazuri compilatorul trebuie să genereze codul corespunzător, fără a “încurca” valorile referite. Se admite şi folosirea unor zone de mărime zero, de exemplu:

`b(3:2)`, primul indice fiind mai mare decât al doilea fără ca să fie precizat un pas negativ. Se admite folosirea tablourilor unidimensionale (vectorilor) pe post de indici de tablouri. Pentru a crea şi atribui valori tablourilor unidimensionale (şi tablourilor de constante) se pot utiliza şi constructori de tablouri, alcătuind chiar expresii. Pentru tablouri cu mai multe dimensiuni, unde constructorii nu pot fi utilizați, se recomandă folosirea funcției RESHAPE (al doilea argument al funcției specifică forma dorită). Iată câteva exemple:

```
INTEGER :: lista(2,3) = &            ! folosirea functiei RESHAPE:
RESHAPE( (/ 11, 12, 21, 22, 31, 32 /), (/2,3/))
!...
zona = (/ 3.14, x, 2.33, y, 3.51 /)  ! utilizarea constructorilor,
indice = (/ (REAL(i), i = 1,10) /)   ! ciclu implicit in cadrul
                                     ! constructorului de tablou,
              ! exemplu cu folosirea unui vector pe post de indice
INTEGER :: vector(4)         ! vectorul ce va fi folosit ca indice
REAL :: tabel(100)           ! tabloul declarat
vector = (/ 33, 11, 12, 13 /) ! initializarea vectorului (indicelor)
WRITE(*,*) tabel(vector)     ! se vor scrie doar elementele de pe
                             ! pozitiile 33, 11, 12 si 13 din
                             ! tabloul ”tabel”.
```

Indicii sub formă de vectori pot fi utilizați în partea stângă a unei asignări/atribuiri doar dacă nu apar valori repetate în lista cu indici (altfel un element ar trebui să fie asociat la două valori diferite). O secțiune de tablou cu indice sub formă de vector nu poate fi țintă într-o instrucțiune de atribuire cu *pointeri* (a se vedea subcapitolul 4.10).

În Fortran 90 există o serie de funcții intrinseci noi, dintre acestea prezentăm sub formă tabelară cele referitoare la tablouri:

Tabel cu funcții intrinseci noi pentru tratarea tablourilor de date:

Funcție	Efect
`ALL(şablon [, DIM=d])`	Are valoarea `.TRUE.` dacă toate elementele filtrate cu *şablon* au această valoare.
`ANY(şablon [, DIM=d])`	Are valoare `.TRUE.` dacă cel puțin un element filtrat cu *şablon* are această valoare.
`COUNT(şablon [, DIM=d])`	Returnează numărul elementelor cu valoarea `.TRUE.` din lista generată de *şablon*.
`CSHIFT(tablou, pas [, DIM=d])`	Translatare circulară a elementelor din tablou (cu pasul *pas*).
`EOSHIFT(tablou, pas [, DIM=d])`	Translatare liniară (End-Off) a elementelor.
`MATMUL(mata, matb)`	Produsul matriceal dintre *mata* şi *matb*.
`MAXLOC(tablou[, şablon])`	Locația (poziția) elementului cel mai mare.
`MINLOC(tablou[, şablon])`	Locația (poziția) elementului cel mai mic.
`MAXVAL(tablou[, DIM=d, şablon])`	Valoarea cea mai mare din tablou.
`MINVAL(tablou[, DIM=d, şablon])`	Valoarea cea mai mică din tablou.

Funcție	Efect
`MERGE(sursaT, sursaF, şablon)`	Combină tablourile conform şablonului (utilizând elementele corespunzătoare din *sursaT* pentru valorile `.TRUE.`, şi din *sursaF* pentru valorile `.FALSE.` din *şablon*).
`PACK(tablou, şablon [,rezultat])`	Împachetează elementele tabloului conform şablonului pasându-le tabloului *rezultat*.
`PRODUCT(tablou[, DIM=d, şablon])`	Produsul valorii elementelor din tablou.
`SUM(tablou[, DIM=d, şablon])`	Suma valorii elementelor din tablou.
`TRANSPOSE(matrice)`	Transpusa matricei (a tabloului bidimensional *matrice*).

Mențiune: funcțiile MAXLOC şi MINLOC aplicate pe un vector vor returna un tablou cu un element, ceea ce nu trebuie confundat cu un scalar (nu sunt identice).

Iată şi câteva exemple pentru ilustrarea funcțiilor prezentate:

```
IF( ANY( a /= b)) THEN ...    ! compararea tablourilor a si b pentru
                              ! a vedea daca sunt identice (conditia
                              ! fiind pusa sub forma: DACA ORICE
                              ! element din A DIFERA fata de elemen-
                              ! tul corespunzator din B ATUNCI...
! ... un alt exemplu:
REAL :: tablou(2,3)           ! declararea unui tablou de forma
tablou=(/ 1, 3, 5, 2, 4, 6 /) !      1 3 5
                              !      2 4 6
! si folosirea functiei SUM va returna:
SUM(tablou)                   ! suma tuturor elementelor: 21
SUM(tablou, DIM=1)            ! suma dupa prima dimensiune,
                              ! adica pe randuri: (/ 9, 12 /)
SUM(tablou, DIM=2)            ! suma dupa a doua dimensiune,
                              ! adica pe coloane: (/ 3, 7, 11 /)

! si inca un exemplu, pentru
! calculul variantzei intr-un tablou, ignorand elementele nule:
media = SUM(x, MASK= x /= 0.0) / COUNT(x /= 0.0)
variantza = SUM((x-media)**2, MASK= x /= 0.0) / COUNT(x /= 0.0)
```

Când unele elemente ale unui tablou trebuie tratate diferit, utilizarea structurii WHERE poate fi foarte utilă, iată de exemplu:

```
WHERE(x /= 0.0)
inversa = 1.0 / x
ELSEWHERE
inversa = 0.0
END WHERE
```

Există şi o formă pe o singură linie, iată de exemplu:

```
WHERE(tablou > 100.0) tablou = 0.0
```

4.4 ALOCAREA DINAMICĂ A MEMORIEI

Există trei modalităţi distincte pentru alocarea dinamică a memoriei la tablouri: automat, prin declarare de tablou alocabil (ALLOCATABLE) şi tablou pointer.

Tablourile automate sunt considerate ca variabile locale, find permise doar în funcţii şi subprograme (declarate corespunzător). Limitele lor sunt stabilite în momentul în care se face apel sau referire la modulul în care au fost declarate. Iată un exemplu:

```
SUBROUTINE finete(npct, spectrum)
IMPLICIT NONE
INTEGER, INTENT(IN) :: npct
REAL, INTENT(INOUT) :: spectrum
REAL :: zona(npct), mai_mare(2*npct)         ! tablouri automate
```

Limitele dimensiunilor pot fi şi expresii de tip întreg, alcătuite din variabile definite şi accesibile la momentul respectiv. În cadrul procedurii tablourile astfel definite se comportă în general la fel cu celelalte tablouri, pot fi utilizate şi de către procedurile subordonate incluse, dar în momentul în care controlul revine la nivelul procedurii gazdă tablourile devin nedefinite. Un tablou automat nu poate fi iniţializat şi nu poate fi folosit nici pentru a salva valori de la un apel la altul. Cele mai multe sisteme stochează asemenea tablouri în stivă (unele sisteme Unix nu alocă prea mult spaţiu pentru stivă).

Tablourile alocabile sunt mai des utilizate deoarece dimensiunile lor se pot fixa în orice moment (implicit încep cu 1), doar rangul (numărul dimensiunilor) trebuie declarat în avans (marcând cu caracterul **:** fiecare dimensiune):

```
REAL, ALLOCATABLE :: vector(:), matrice(:,:), trei_d(:,:,:)
!...
ALLOCATE(vector(123), matrice(0:9,-2:7))    ! declararea dimensiunii
!...
DEALLOCATE(matrice, vector)                 ! eliberarea memoriei
```

Tablourile alocabile pot fi pasate procedurilor subordonate fără probleme, însă înainte de terminarea procedurii în care au fost declarate ele trebuie dealocate (folosind instrucţiunea DEALLOCATE, ca în exemplul de mai sus) pentru a evita problemele de "scurgeri" de memorie. Dacă unui asememnea tablou i s-au fixat dimensiunile, acestea nu pot fi modificate (doar printr-o nouă alocare după o dealocare prealabilă). Nu se admite alocarea dublă a aceluiaşi tablou, se poate folosi funcţia intrinsecă ALLOCATED pentru a evita asemenea situaţii:

```
IF(ALLOCATED(tablou)) THEN           ! utilizarea functiei
DEALLOCATE(tablou)                   ! dealocare daca se impune
END IF
ALLOCATE(tablou(1:nouadimensiune))   ! realocare
```

Pentru verificarea succesului alocării spaţiului în cazul unor asemenea tablouri, cu dimensiuni extinse, se poate apela la interogarea unei variabile de stare care în mod normal returnează valoare nulă, iar în cazul unei erori de alocare va returna o valoare diferită:

```
ALLOCATE(tablou_imens(1:npct), STAT=ierror)
IF(ierror /= 0) THEN
WRITE(*,*)"Eroare la incercarea de alocare pentru tablou_imens "
STOP
END IF
```

În cazul unor asemenea erori se recomandă aplicarea unui alt algoritm, cu nevoi mai reduse de memorie dacă e posibil, altfel programul se va termina neaşteptat.

Un tablou alocabil nu poate fi specificat într-o instrucțiune COMMON, DATA, EQUIVALENCE sau NAMELIST. Poate însă avea atributul SAVE, ceea ce-i furnizează un caracter global în cadrul modulului.

Tablourile pointer sunt a treia modalitate de alocare dinamică. Un pointer nu conține date, dar indică către un scalar sau un tablou în care sunt stocate date. Poate fi considerat deci, ca o referință către o referință. Dată fiind natura lui, nu are spațiu de stocare de la început alocat, ci doar la execuția programului. Un tablou alocabil nu poate fi transmis unei proceduri dacă nu a fost alocat în prealabil, dar cu un tablou pointer se poate realiza acest lucru. Iată un exemplu:

```
PROGRAM pdemo
IMPLICIT NONE
REAL, POINTER :: ptablou(:)          ! declararea unui tablou pointer
OPEN(UNIT=1, FILE=’fisier’, STATUS=’old’)
CALL citire(1, ptablou)
WRITE(*,*)’tablou de ’, SIZE(tablou), ’ puncte:’
WRITE(*,*) ptablou
DEALLOCATE(ptablou)
STOP                                 ! STOP-ul este optional
CONTAINS
 SUBROUTINE citire(unitate, x)
 INTEGER, INTENT(IN) :: unitate
 REAL, POINTER :: x(:)               ! pointer nu poate avea INTENT
 INTEGER :: npuncte
 READ(unitate) npuncte               ! numarul punctelor de citit
 ALLOCATE(x(1:npuncte))              ! alocarea spatiului
 READ(unitate) x                     ! citirea intregului tablou
 END SUBROUTINE citire
END PROGRAM pdemo
```

Se poate observa din exemplul de mai sus, că la declararea tablourilor pointer se aplică o sintaxă similară cu cea de la tablourile alocabile (se marchează cu caracterul **:** pozițiile dimensiunilor sau, altfel spus, rangul tabloului). Exemplul prezentat este simplu pentru că prezintă o procedură internă, deci compilatorul se va descurca foarte uşor cunoscând toate detaliile interfeței la traducerea apelului subprogramului (la transmiterea unui pointer către o procedură, se cere o asemenea “interfață explicită”, noțiune ce va fi prezentată în subcapitolul următor).

4.5 MODULE ŞI INTERFEȚE

În Fortran 90 există patru tipuri de unități de program:

1. *Programul principal* (Main – care începe de regulă cu declarația PROGRAM).
2. *Proceduri externe* sau *subprograme* (ce încep fie cu declarația SUBROUTINE, fie cu FUNCTION).
3. Unități cu *blocuri de date* (specificați prin declarația BLOCK DATA).
4. *Module* (specificați prin declarația MODULE) ce pot conține orice combinație a următoarelor elemente:
   - definiții de constante;
   - definiții de date derivate (structuri de date);
   - declarații de stocare;
   - proceduri (subprograme şi funcții).

Un modul poate fi accesat în orice unitate de program prin instrucțiunea USE (inclusiv dintr-un alt modul). Instrucțiunea USE trebuie să preceadă celelalte declarații, ea trebuie să fie prima după specificația de identificare a unității de program. Deşi modulul poate fi conținut într-un fişier separat sau în aceeaşi fişier cu celelalte unități de program, el trebuie compilat înainte de unitatea care conține referința la el. Cele mai multe compilatoare suportă compilarea separată a modulelor, generând fişiere cu extensia ".mod" (sau ceva similar) din ele. Din acest punct de vedere, utilizarea modulelor prin instrucțiunea USE s-ar asemăna cu inserarea prin instrucțiunea INCLUDE, dar de fapt utilizarea modulelor este o facilitate mai performantă din cauza procedurilor de modul.

Un modul începe de regulă cu secțiunea datelor urmată de o specificație CONTAINS (dacă conține vreo procedură), după care urmează descrierile procedurilor de modul. Aceste proceduri de modul au acces direct la toate definițiile de date şi specificații de stocare prin asociere. Permit încapsularea datelor şi a mulțimii de proceduri ce operează cu aceste date sau ce folosesc zona de stocare pentru schimb de valori (intercomunicare). Următorul exemplu prezintă un modul ce tratează ieşirea pe un videoterminal sau într-o fereastră de tipul *X-term* (prin secvențe *escape*, similare cu cele din ANSI.SYS sub DOS):

```
MODULE vt_mod                           ! specificarea modulului
 IMPLICIT NONE                          ! valabil intregului modul
 CHARACTER(1), PARAMETER :: escape = achar(27)
 INTEGER, SAVE :: latime_ecran = 80, inaltime_ecran = 24
CONTAINS
 SUBROUTINE goleste                     ! Goleste ecranul si muta
                                        ! cursorul in stanga sus
  CALL afiseaza( escape // "[H" // escape // "[2J")
 END SUBROUTINE goleste

 SUBROUTINE fa_latime(latime)           ! seteaza noua latime de ecran
  INTEGER, INTENT(IN) :: latime         ! la latimea dorita (80 sau 132)
  IF(latime > 80) THEN
   CALL afiseaza( escape // "[?3h" ) ! comutare la 132 de coloane
   latime_ecran = 132
  ELSE
```

```
    CALL afiseaza( escape // "[?3l" ) ! comutare la 80 de coloane
    latime_ecran = 80
   END IF
  END SUBROUTINE fa_latime

  SUBROUTINE ia_latime(latime)         ! returneaza latimea ecranului
   INTEGER, INTENT(OUT) :: latime      ! (de 80 sau 132 de caractere)
   latime = latime_ecran
  END SUBROUTINE ia_latime

  SUBROUTINE afiseaza(rand)            ! doar pentru uz intern
   INTEGER, INTENT(IN) :: rand
   WRITE(*, "(1X,A)", ADVANCE="NO") rand
  END SUBROUTINE afiseaza
END MODULE vt_mod
```

Pentru a folosi acest modul, la începutul sursei trebuie inserat doar:

```
USE vt_mod
```

după ce exemplul de mai sus a fost compilat, fiind transformat într-un fişier imagine obiect (cu extensia *.mod*).

Toate variabilele din modul sunt accesibile în mod implicit tuturor programelor care utilizează modulul. Acest aspect însă poate fi deranjant: de multe ori, procedurile de modul oferind toate funcțiunile necesare de acces, este preferabil ca utilizatorii să nu se amestece cu aspectele interne. Pentru a schimba accesibilitatea generică de tip PUBLIC a numelor simbolice dintr-un modul se poate utiliza specificația PRIVATE în cadrul acestuia. Uneori însă această soluție nu este suficientă: se poate întâmpla ca un modul să conțină nume de proceduri sau de variabile care să intre în conflict cu cele alese deja de un utilizator. Pentru asemenea situații există două soluții. Dacă nu se doreşte utilizarea întregului modul, din motivul evitării conflictului dintre unele nume folosite, se poate specifica o listă conținând doar entitățile publice sau identificatorii generici doriți a fi utilizați din modul, prin combinația USE ONLY, ca de exemplu:

```
USE vt_mod, ONLY: goleste
```

Cealaltă soluție ar fi redenumirea entității al cărei nume deranjează (crearea unui *alias* pentru numele simbolic al acesteia). De exemplu, dacă numele *ia_latime* din modulul prezentat mai sus ne creează probleme, putem proceda în felul următor:

```
USE vt_mod, e_lata => ia_latime
```

şi astfel numele *e_lata* va înlocui temporar numele *ia_latime* în cadrul modulului *vt_mod*.

Modulele permit încapsularea unei structuri de date împreună cu mulțimea procedurilor ce-l manipulează, ceea ce în programarea orientată obiect înseamnă că ele pot conține o *clasă* şi *metodele* acesteia.

Atunci când o procedură de metodă este apelată, se spune că este vorba de o *interfață explicită*, ceea ce înseamnă că compilatorul poate verifica consistența argumentelor actuale efective şi fictive. Aceasta se consideră o trăsătură foarte valoroasă deoarece în Fortran 77

asemenea interfețe nu se puteau verifica şi erorile erau obişnuite din această cauză. Din acest punct de vedere, utilizarea modulelor se dovedeşte mai avantajoasă decât folosirea blocurilor comune (COMMON), a blocurilor de date (BLOCK DATA) sau a punctelor de intrare (ENTRY).

Modulele furnizează un nivel structural suplimentar în proiectarea programelor, putând clasifica nivelurile în ordine descrescătoare în felul următor: *programe*; *module*; *proceduri*; *instrucțiuni*.

Interfețele explicite derivate din module permit utilizarea multor facilități avansate, cum ar fi: tablourile de pointeri cu formă presupusă, argumententele opționale, operatorii definiți de către utilizator, numele generice etc. Există însă şi câteva dezavantaje:

- fiecare modul trebuie compilat înainte de unitățile de program apelante;
- necesită atenție mărită;
- scrise într-un singur fişier, modulele trebuie să preceadă programul principal;
- dacă un modul este modificat, vor trebui recompilate toate unitățile de program apelante ceea ce poate conduce la consum excesiv de timp;
- un modul rezultă de regulă într-un singur fişier imagine obiect, reducând avantajele bibliotecilor imagine obiect, putând conduce la fişiere executabile de dimensiuni mari.

O interfață explicită înseamnă că argumentele fictive ale procedurii sunt vizibile compilatorului în momentul traducerii instanței de apel către procedura respectivă. O interfață se consideră explicită atunci când:

- se apelează o procedură de modul de către o unitate de program sau de către o altă procedură din cadrul aceluiaşi modul;
- se apelează o procedură internă de către o unitate gazdă sau de către orice altă procedură din aceeaşi gazdă;
- se apelează o funcție intrinsecă (internă);
- se specifică un bloc de interfață explicită;
- o procedură recursivă se autoapelează direct sau indirect.

Specificarea unu bloc de interfață are următoarea formă:

```
INTERFACE [specificator_generic]
  [corp interfaţă]
  [MODULE PROCEDURE listă_nume]
END INTERFACE [specificator_generic]
```

Un bloc de interfață poate fi specificat şi în cadrul unui modul, pentru a facilita utilizarea acestuia. În cazul în care se folosesc biblioteci existente din Fortran 77, se recomandă crearea unui modul care să conțină toate interfețele de proceduri (există chiar programe de conversie şi generare pentru asemenea situații).

4.6 PROCEDURI ÎN FORTRAN 90

În cazul folosirii tablourilor pe post de parametri transferați între proceduri, se recomandă utilizarea tablourilor cu formă presupusă. Un asemenea tablou este un argument formal care își asumă mărimea tabloului din postura argumentului efectiv, însă rangul (şi implicit dimensiunile) poate să fie diferit față de tabloul asociat (din postura argumentului efectiv). De aici rezultă că la specificarea unui asemenea tablou (cu formă presupusă) se specifică rangul, marcând prin caracterul **:** fiecare dimensiune. Forma efectivă va fi luată de fiecare dată doar în momentul realizării apelului (prin argumentele efective). Limita inferioară a indicilor este implicit 1, ea nu trebuie să corespundă neapărat cu cea a tablourilor efective din apel, ca atare funcțiile intrinseci LBOUND şi UBOUND nu vor putea servi informații adiționale.

Dacă există o interfață implicită, se pot folosi cuvinte cheie în loc de notațiile poziționale uzuale. De asemenea, toate funcțiile intrinseci pot fi apelate prin cuvinte cheie. Utilizarea apelurilor prin cuvinte cheie se poate dovedi utilă când argumentele opționale se omit:

```
INTEGER :: vector(8)
CALL DATE_AND_TIME(VALUES=vector)
```

În cadrul unui apel se pot amesteca cuvintele cheie cu argumente poziționale, dar acestea din urmă trebuie să fie în față. Argumente opționale pot fi furnizate în proceduri scrise de utilizator, însă testarea existenței lor (prin funcția intrinsecă PRESENT) este esențială înainte de utilizarea lor (cu excepția cazului unui alt apel al unei proceduri cu un argument opțional):

```
SUBROUTINE scrie_text(sir, ngol)
CHARACTER(*), INTENT(IN) :: sir                ! linie de text,
INTEGER, INTENT(IN), OPTIONAL :: ngol          ! linii goale inainte,
INTEGER :: localgol                            ! stocare locala
IF(PRESENT(ngol)) then
  localgol = ngol
ELSE
  localgol = 0                                 ! valoare implicita,
END IF
! ...
END SUBROUTINE scrie_text
```

Apelurile posibile din cadrul diverselor unități de program, către subprogramul de mai sus, pot arăta în felul următor:

```
CALL scrie_text("titlu document")             ! al doilea argument omis
CALL scrie_text("22 Decembrie 1989", 3)
CALL scrie_text(ngol=5, sir="un alt rand")    ! in alta ordine
```

Argumentele opționale de la sfârşitul listei de parametri pot fi omise la apel fără probleme, dar dacă se omit argumentele din cadrul listei, atunci trebuie folosite cuvinte cheie pentru cele următoare (nu este suficientă utilizarea unor virgule succesive ca în anumite extensii ale limbajului Fortran 77).

Funcțiile intrinseci au în general nume generice, astfel o funcție poate furniza valori diferite în funcție de tipul argumentelor utilizate. Funcțiile definite de utilizator pot avea la fel, nume generice. Presupunând că am scrie un modul în care să avem mai multe funcții similare, de exemplu pentru sortare: *sortare_i* pentru valori întregi, *sortare_r* pentru valori reale şi *sortare_s* pentru şiruri de caractere, numele generic *sortare* s-ar putea defini în felul următor:

```
INTERFACE sortare
 MODULE PROCEDURE sortare_i, sortare_r, sortare_s
END INTERFACE
```

Regulile pentru rezolvarea numelor generice sunt destul de complicate, însă este suficient să ne asigurăm ca fiecare procedură să difere de celelalte care au acelaşi nume prin tipurile valorilor, rangul tablourilor sau cel puțin printr-un argument utilizat efectiv.

Procedurile se pot autoapela în mod direct sau indirect dacă sunt declarate recursive (prin specificația RECURSIVE). Cazurile tipice se întâlnesc la tratarea structurilor de date similare cum ar fi arbori de directoare, de tip B etc. Un exemplu clasic este calculul factorialului unui număr întreg:

```
RECURSIVE FUNCTION factorial(n) RESULT(nfact)
IMPLICIT NONE
INTEGER, INTENT(IN) :: n
INTEGER :: nfact
IF(n > 0) THEN
  nfact = n * factorial(n-1)
ELSE
  nfact = 1
END IF
END FUNCTION factorial
```

La fel de simplu se poate realiza exemplul arătat şi utilizând un ciclu DO. Folosirea variabilei RESULT este opțională, ea fiind necesară doar pentru a evita ambiguitatea sintaxei (de exemplu, dacă funcția ar returna un tablou, un element de tablou nu s-ar distinge de un apel de funcție).

4.7 STRUCTURI DE DATE, TIPURI DERIVATE

Noțiunile de *date definite de utilizator*, *structuri de date* şi *tipuri derivate de date* înseamnă toate unul şi acelaşi lucru. Instrucțiunea declarativă pentru tipuri derivate (TYPE) permite specificarea proprietăților obiectelor şi funcțiilor definite de utilizator. Iată un exemplu simplu, ilustrând tratarea unei liste de puncte:

```
TYPE :: tip_punct
 CHARACTER(10) :: nume              ! numele obiectului
 REAL :: x, y, z                    ! coordonatele punctului
 INTEGER :: culoare                 ! culoarea punctului
END TYPE tip_punct
```

După cum se poate observa, în asemenea structuri se pot amesteca şi entităţi de tip caracter cu entităţi numerice şi logice (spre deosebire de blocurile comune), ajustarea aspectului fizic pentru un acces eficient căzând în sarcina compilatorului. Exemplul de mai sus descrie doar structura dorită, iar pentru a creea variabile cu tipul definit de date de mai sus instrucţiunea TYPE trebuie folosită într-o altă formă, şi anume:

```
TYPE(tip_punct) :: punct_vechi, noua_lista(20)
```

Prin această specificare s-a creat o variabilă structurată (sub numele: *punct_vechi*) şi un tablou cu 20 de elemente (*noua_lista*), fiecare dintre ele având cele cinci componente specificate (*nume*, *x*, *y*, *z* şi *culoare*). Bineînţeles, tipul derivat referit trebuie să fie definit înainte de specificarea unor obiecte de acel tip.

Componentele unei structuri astfel definite sunt accesate utilizând caracterul **%** (şi nu punctul ca în cele mai multe limbaje de programare). Astfel *punct_vechi%nume* este o variabilă de tip caracter, iar *noua_lista(11)%x* este o variabilă de tip real. Asemenea entităţi structurale pot fi folosite în mod similar cu variabilele simple, de exemplu:

```
! ...
noua_lista(1)%nume = "primul"
noua_lista(1)%x = 23.4
noua_lista(1)%y = -2.15
noua_lista(1)%z = 0.0
noua_lista(1)%culoare = 8
! ...
noua_lista(2) = punct_vechi          ! copierea tuturor componentelor
noua_lista(2)%x = noua_lista(2)%x + 1.5
! ...
```

Numele componentelor sunt considerate locale în cadrul structurii, deci nu apar probleme dacă aceeaşi unitate de program foloseşte variabile simple cu denumiri similare (ca *nume*, *x* etc. dacă considerăm exemplul prezentat).

Constructorii de structură permit specificarea valorii obiectelor de tip derivat, numele tipului este folosit ca şi cum ar fi vorba de o funcţie de conversie, cu o listă de valori ale componentelor pe post de argumente:

```
noua_lista(3) = tip_punct("al treilea", 43.21, -15.3, 0.0, 2)
```

Dacă există un tablou declarat ca şi tip derivat, fiecare componentă a structurii poate fi tratată ca tablou (conform exemplului prezentat *noua_lista%x* va fi un tablou cu 20 de valori de tip real). Elementele acestor entităţi nu se pot situa în locaţii adiacente în memorie, de acest aspect se va îngriji compilatorul.

Variabilele structurate pot fi utilizate şi în instrucţiuni de intrare/ieşire. În cazul unor operaţii de citire/scriere fără format sau cu format implicit nu se pune nici o problemă, însă în cazul operaţiilor formatate trebuie alcătuită o listă corespunzătoare de descriptori:

```
WRITE(*,*) punct_vechi                      ! cu format implicit
READ(fisier, "(A,3F5.2,I2)") noua_lista(4)  ! cu format explicit
```

De asemenea, putem defini structuri înlănțuite, ca în exemplul următor:

```
TYPE :: punct
 REAL :: x, y                  ! coordonatele punctului
END TYPE punct

TYPE :: linie
 TYPE(punct) :: capat(2)       ! coordonatele capetelor
 INTEGER :: grosime            ! grosimea liniei in pixeli
END TYPE linie

TYPE(line) :: v
REAL :: lungime
v = linie( (/ punct(1.2,2.4), punct(3.5,7.9) /), 2)
lungime = SQRT((v%capat(1)%x - v%capat(2)%x)**2 &
+ (v%capat(1)%y - v%capat(1)%y)**2)
! ...
```

O limitare a acestor structuri în Fortran este necesitatea fixării în prealabil a lungimii componentelor de tip tablou, adică, altfel spus un tablou alocabil nu poate fi componenta unei structuri definite de utilizator. Din fericire însă, se admit pointeri ca şi componente:

```
TYPE :: tip_document
CHARACTER(80), POINTER :: rand(:)    ! componenta tip pointer
END TYPE tip_document
! ...
TYPE(tip_document) :: doc     ! declara o variabila structurata
ALLOCATE(doc%rand(200))       ! spatiu alocat pentru 200 de randuri
```

Pentru a face structura şi mai flexibilă am putea aloca un tablou de variabile de tip caracter cu lungimea de câte un caracter a elementelor, pentru fiecare rând (deşi probabil ar fi mai greu de utilizat aşa). Pentru a transmite o variabilă structurată într-o procedură, pe ambele părți ale interfeței trebuie să asigurăm aceeaşi definiție de structură. Cea mai simplă cale pentru acest deziderat este folosirea unui modul. Există însă două limitări în ceea ce priveşte utilizarea tipurilor derivate de date care conțin pointeri:

- ele nu se pot utiliza în listele de intrare/ieşire ale instrucțiunilor de citire/scriere;
- dacă o instrucțiune de asignare copiază valorile unei date derivate într-alta, deşi orice componentă de tip pointer va fi copiată, noul pointer va indica tot aceeaşi zonă de memorie.

Atunci când se defineşte un nou tip de date, ar fi de dorit ca obiectele de acel tip să se poată utiliza în expresii obişnuite. Evident, ar fi mai simplu să scriem `a*b+c*d` decât `add(mult(a,b),mult(c,d))`. În acest scop însă, fiecare operator pe care dorim să-l utilizăm va trebui definit (sau *supraîncărcat*) pentru fiecare tip derivat de date. Exemplul următor prezintă definirea unei structuri de date cu numele *fuzzy*, care conține un număr real şi eroarea sa standard. Când două asemenea valori *fuzzy* sunt adunate, erorile se vor aduna la pătrat. Pentru o astfel de adunare am supraîncărcat operatorul "+":

```
MODULE fuzzy_mate
IMPLICIT NONE
```

```
TYPE fuzzy
 REAL :: valoare, eroare
END TYPE fuzzy

INTERFACE OPERATOR (+)               ! interfata explicita
 MODULE PROCEDURE fuzzy_plus_fuzzy
END INTERFACE

CONTAINS
 FUNCTION fuzzy_plus_fuzzy(prim, secund) RESULT (suma)
 TYPE(fuzzy), INTENT(IN) :: prim, secund    ! INTENT necesar
 TYPE(fuzzy), INTENT(OUT) :: suma           ! INTENT opţional
 sum%valoare = prim%valoare + secund%valoare
 sum%eroare = SQRT(prim%eroare**2 + secund%eroare**2)
 END FUNCTION fuzzy_plus_fuzzy
END MODULE fuzzy_mate

PROGRAM test_fuzzy
IMPLICIT NONE
USE fuzzy_mate
TYPE(fuzzy) a, b, c
a = fuzzy(10.0, 4.0) ; b = fuzzy(12.5, 3.0)
c = a + b
PRINT *, c
END PROGRAM test_fuzzy
```

Rezultatul afişat de programul de test ar trebui să fie: `22.5 5.0`.

Şi operatorul "=" (instrucţiunea de asignare) poate fi supraîncărcat pentru tipuri derivate de date, însă în cazul de mai sus nu a fost necesar pentru că am folosit un subprogram cu un argument având atributul INTENT(IN) şi celălalt INTENT(OUT).

O implementare completă a clasei *fuzzy* ar trebui să cuprindă supraîncărcarea operatorilor aritmetici, funcţii intrinseci (ca de exemplu: SQRT), respectiv combinaţii între operanzi de tip real şi *fuzzy*. Ulterior, reprezentarea internă poate fi schimbată fără a afecta programele care folosesc tipul derivat *fuzzy*, cu condiţia ca interfeţele să rămână nemodificate.

Prioritatea la evaluare a unui operator existent rămâne neschimbată prin supraîncărcare, iar operatorii unari noi vor avea o prioritate mai mare decât cei intrinseci existenţi, pe când noii operatori binari vor avea prioritate mai scăzută. La supraîncărcarea unui operator existent se recomandă ca semnificaţia lui să nu fie alterată, altfel e mai sănătos să se inventeze unul nou pentru operaţia dorită.

4.8 INTRĂRI ŞI IEŞIRI

Intrările şi ieşirile cuprind aspectele legate de transferul datelor, precum şi cele referitoare la conectarea, interogarea, modificarea, şi poziţionarea fişierelor. Ca şi în cazul limbajului Fortran 77, există fişiere considerate externe (într-un mediu extern programului executabil) şi fişiere considerate interne (în spaţiul de stocare al memoriei interne). Din punctul de

vedere al operațiilor posibile de citire şi scriere, articolele (înregistrările) din cadrul fişierelor pot fi considerate de trei tipuri: formatate, neformatate, şi marcajul de sfârşit de fişier (EOF, poate fi scris într-un fişier secvențial printr-o instrucțiune ENDFILE). Fiecărui tip de articol îi corespunde o modalitate de citire. Înregistrările formatate se citesc prin instrucțiuni cu format explicit sau implicit, iar cele neformatate prin instrucțiuni fără format. Articolele neformatate pot să fie goale (să nu conțină date), reprezentarea internă a datelor din asemenea înregistrări fiind dependentă de echipamentul de calcul.

În ceea ce priveşte accesul la fişiere, prin dezvoltarea limbajului în varianta 90 există câteva parametri şi opțiuni noi pentru deschiderea acestora, iată-le în tabelul următor:

Tabel cu parametrii noi din instrucțiunea OPEN:

Cuvântul cheie	Valoarea	Funcțiunea
ACTION=	"READ"	Acces doar pentru citire.
	"WRITE"	Acces pentru scriere.
	"READWRITE"	Acces pentru citire şi scriere.
POSITION=	"APPEND"	Adăugarea la un fişier secvențial existent.
	"REWIND"	Asigurarea poziționări la începutul fişierului deschis.
STATUS=	"REPLACE"	Suprascrierea unui fişier existent, sau crearea unui fişier nou.
RECL=	*lungime*	Se poate folosi la crearea unui fişier secvențial în scopul precizării lungimii maxime a înregistrărilor.

Valoarea *lungime* a parametrului RECL semnifică numărul de caractere în cazul accesului formatat, iar în cazul accesării neformatate (acces direct, binar) semnificația este în funcție de sistem. Instrucțiunea INQUIRE beneficiază de asemenea de cuvinte cheie adiționale pentru a returna informații legate de deschiderea unei unități. De exemplu, precizând un specimen pentru o listă de intrare/ieşire, ne poate returna valoarea lungimii necesare parametrului RECL din instrucțiunea OPEN:

```
INQUIRE(IOLENGTH=lungime) specimen, lista, de, elemente
OPEN(UNIT=unitate, FILE=specfis, STATUS="new", &
ACCESS="direct", RECL=lungime)
```

Se pot utiliza citiri şi scrieri dirijate prin liste (cu format implicit/liber) şi asupra unor fişiere interne, iată cum:

```
CHARACTER(LEN=10) :: sir
sir = " 3.14 "
READ(sir, *) variabila
```

De asemenea, pe lângă descriptorii deja cunoscuți sunt disponibile variante mai performante şi chiar noi, pentru formatul operațiilor de intrare/ieşire, prezentate în tabelul următor.

Tabel complementar, cu descriptori noi:

Tip	Sintaxă	Efect
B	`[n]Bw[.z]`	Pentru transferul valorilor sub formă binară. Prin *z* se poate specifica numărul zerourilor premergătoare la afişare.
ES	`[n]ESw.d[Ee]`	Formă exponențială (notație "ştiințifică"), cu zecimale după prima cifră semnificativă (exemplu: 1.234E-01 în loc de 0.1234E-00).
EN	`[n]ENw.d[Ee]`	Formă exponențială (notație "tehnică"), exponentul fiind multiplu de 3 (exemplu: 123.4E-03 în loc de 0.1234E-00).
G	`[n]Gw.d[Ee]`	Descriptor modificat, cu caracter general, poate fi utilizat atât pentru valori numerice cât şi pentru valori logice sau de tip caracter.
O	`[n]Ow[.z]`	Pentru transferul valorilor sub formă octală. Prin *z* se poate specifica numărul zerourilor premergătoare la afişare.
P	`mP`	Interpretează anumite numere reale cu un factor de mărime *m* specificat.
Q	`Q`	Returnează numărul caracterelor rămase într-o înregistrare citită.
Z	`[n]Zw[.z]`	Pentru transferul valorilor sub formă hexazecimală. Prin *z* se poate specifica numărul zerourilor premergătoare la afişare.
$	`$`	Inhibă returul de car la operația curentă de intrare/ieşire (similar cu descriptorul "\" din Fortran 77).
:	`:`	Încheie controlul formatării (dacă nu mai sunt elemente în listă).

Operațiile de citire/scriere tratează în mod normal o înregistrare completă. Există însă şi o facilitate nouă de a trata înregistrările, prin specificarea atributului ADVANCE în cadrul acestor instrucțiuni, ca în exemplul următor:

```
WRITE(*, "(A)", ADVANCE="NO") "Numarul ciclurilor: "
READ(*,*) n_ciclu
```

În acest caz, în loc de trecerea la articolul următor se va muta doar un pointer teoretic în cadrul articolului atât cât este necesar, permițând introducerea valorii variabilei *n_ciclu* pe linia curentă pe care s-a afişat mesajul (ca la un prompter) pe ecran. Acest atribut combinat cu atributul SIZE permite şi măsurarea lungimii actuale a rândului citit:

```
CHARACTER(LEN=80) :: text
INTEGER :: n_car
READ(*, "(A)", ADVANCE="no", SIZE=n_car) text
```

Această facilitate se poate dovedi utilă în cazul citirii unor fişiere secvențiale parțial corupte sau cu structuri neconvenționale.

4.9 TRATAREA ŞIRURILOR DE CARACTERE

Aşa cum am mai menţionat, există o sumedenie de funcţii intrinseci noi şi performante introduse prin Fortran 90, dintre acestea unele fiind destinate tratării şirurilor sau a subşirurilor alcătuite din caractere. În tabelul următor se pot vedea aceste funcţii, împreună cu descrierea efectului produs prin apelarea lor.

Tabel cu noile funcţii intrinseci pentru tratarea şirurilor de caractere:

Funcţie		Efect
Nume generic	Tip	
`ACHAR(n)`	CH	Returnează caracterul de pe poziţia *n* din tabela ASCII.
`IACHAR(c)`	I	Returnează poziţia caracterului *c* din tabela ASCII.
`LEN_TRIM(şir)`	I	Returnează lungimea lui *şir* (în număr de caractere) ignorând spaţiile de la sfârşit.
`TRIM(şir)`	CH	Returnează *şir*-ul fără spaţiile de la sfârşit.
`ADJUSTL(şir)`	CH	Elimină spaţiile premergătoare din *şir*.
`ADJUSTR(şir)`	CH	Elimină spaţiile finale din *şir*.
`REPEAT(şir, n)`	CH	Concatenare repetată de *n* ori a *şir*-ului.
`INDEX(şir, s[, BACK=v])`	I	Poziţia primului caracter al subşirului *s* în *şir* (dacă *v* este `.TRUE.`, *şir* va fi parcurs dinspre capăt către început).
`SCAN(şir, s[, BACK=v])`	I	Poziţia primului caracter al subşirului *s* în *şir*, sau al ultimului caracter din *s* dacă *v* este `.TRUE.`.
`VERIFY(şir, s[, BACK=v])`	I	Poziţia primului caracter din *şir* care diferă de caracterele din subşirul *s* (dacă *v* este `.TRUE.`, *şir* va fi parcurs dinspre capăt).

Prin noua sintaxă sunt admise şi suprapuneri de subşiruri, ca în exemplul următor:

```
text(1:5) = text(3:7)        ! nepermis in Fortran77, acum admis
```

Operatorul de concatenare "//" poate fi aplicat fără restricţii şi asupra argumentelor (cu lungimea transmisă) din proceduri.

Funcţiile care tratează entităţi de tip caracter pot returna şiruri cu o lungime ce depinde de argumentele specificate, de exemplu:

```
FUNCTION concat(s1, s2)
 IMPLICIT NONE
 CHARACTER(LEN=LEN_TRIM(s1)+LEN_TRIM(s2)) :: concat ! numele functiei
 CHARACTER(LEN=*), INTENT(IN) :: s1, s2
 concat = TRIM(s1) // TRIM(s2)
END FUNCTION concat
```

Se pot folosi şiruri cu lungimea zero, cum ar fi referinţe de genul `subşir(i:j)` unde *i* > *j*, şi constante de forma `""`. Sunt admise şi subşiruri de constante, iată un exemplu pentru convertirea unei valori *n* de tip întreg, din domeniul 0–9, în caracterul corespunzător:

```
caracter_n = "0123456789"(n:n)    ! alocarea cifrei ca si caracter.
```

Variabila *caracter_n* va conţine cifra corespunzătoare valorii lui *n*, cu condiţia ca *n*-ul precizat să nu fie mai mic decât zero sau mai mare decât 9 (altfel va rezulta eroare).

4.10 POINTERI

Foarte multe limbaje de programare suportă pointeri pentru că utilizarea acestora uşurează implementarea structurilor dinamice de date: liste înlănţuite, stive şi arbori. Programele scrise în limbajul C depind în mare măsură de pointeri deoarece transmiterea unui tablou către o funcţie generează instantaneu o referinţă de acest tip. Însă utilizarea pointerilor poate crea şi greutăţi:

- Folosirea lor poate forţa programatorul să ia în considerare aspecte de un nivel apropiat maşinii, ceea ce ar fi mai eficient dacă s-ar lăsa pe seama compilatorului.
- Utilizarea lor excesivă conduce la surse greu inteligibile şi greu mentenabile.
- Conduc foarte uşor la erori detectabile doar în faza de rulare (o mare parte din greşelile sub limbajul C sunt datorate utilizării accidental eronate a pointerilor).
- Inhibă optimizarea automată a codului la compilare (două obiecte aparent distincte se pot dovedi a fi doar pointeri indicând către aceeaşi locaţie de memorie).

Limbajul Java este considerat de către unii ca fiind un dialect fără pointeri al limbajului C++. Deşi pointerii din Fortran sunt relativ “blânzi”, ei trebuie folosiţi totuşi cu atenţie.

După cum am menţionat în subcapitolul referitor la tablouri, un pointer poate fi perceput ca o referinţă către o referinţă. Deci un pointer nu conţine date, el indică doar către o variabilă care conţine date. În Fortran, poate indica doar către un alt pointer sau către o variabilă declarată explicit ca fiind o ţintă validă (prin specificatorul TARGET). Fiecare pointer are o stare de asociere care-l determină să indice sau nu, la un moment dat, către un obiect ţintă.

Funcţia intrinsecă ASSOCIATED permite interogarea acestei stări. Această funcţie va returna valoarea `.FALSE.` în cazul unei stări neasociate (iar în caz contrar valoarea `.TRUE.`).

În Fortran 90, din păcate, la declararea iniţială a unui pointer (prin atributul POINTER) starea acestuia este nedefinită (în Fortran 95 acest aspect este însă rezolvat). Din acest motiv, cea mai bună metodă este ca la început să setăm fiecare pointer într-o stare neasociată (prin instrucţiunea NULLIFY), după care se va putea testa starea lor (apelând funcţia internă ASSOCIATED) înaintea alocării lor unor ţinte valide. Starea de asociere a pointerilor poate fi desetată şi prin instrucţiunea DEALLOCATE (sau prin apelarea funcţiei intrinseci NULL).

Limbajul Fortran 90 nu permite crearea unor tablouri din pointeri, dar permite crearea unor tablouri de obiecte definite de utilizator (de tip derivat) care să conțină pointeri:

```
TYPE :: tablou_de_ptr
 REAL, DIMENSION(:), POINTER :: tp
END TYPE tablou_de_ptr

TYPE(tablou_de_ptr), ALLOCATABLE :: x(:)
!...
ALLOCATE(x(nx))
DO i = 1,nx
 ALLOCATE(x(i)%tp(m))
END DO
```

Pointerilor le pot fi asignate valori de tip țintă, printr-o operație specială marcată cu simbolul => (pentru a se deosebi de operațiile curente de asignare sau de atribuire). Dacă ținta nu este definită sau asociată, pointerul primeşte acelaşi statut ca şi variabila dinamică reprezentată de țintă. Pointerii se pot dovedi foarte utili la notarea prescurtată a secțiunilor de tablouri, ca şi alias, de exemplu:

```
REAL, TARGET :: poza(1024,768)       ! declarare ca tinta
REAL, DIMENSION(:,:), POINTER :: alfa, beta
alfa => poza(1:512, 385:768)         ! asignare la un sfert din poza
beta => poza(1:1024:1, 768:1:-1)     ! asignare la poza oglindita
```

Iată şi ilustrarea unei situații (printr-o funcție de realocare cuprinsă într-un modul) în care se poate dovedi utilă returnarea unui pointer pentru un tablou, de către o funcție:

```
MODULE realocare_m
CONTAINS
 FUNCTION realoca(p, n)                      ! realoca de tip real
  REAL, POINTER, DIMENSION(:) :: p, realoca  ! declarare pointeri
  INTEGER, intent(in) :: n
  INTEGER :: n_vechi, i_eroare
  ALLOCATE(realoca(1:n), STAT=i_eroare)
  IF(i_eroare /= 0) STOP "Eroare de alocare" ! tratarea erorii
  IF(.NOT. ASSOCIATED(p)) RETURN             ! testarea asocierii
  n_vechi = MIN(SIZE(p), n)
  realoca(1:n_vechi) = p(1:n_vechi)          ! atribuire "comuna"
  DEALLOCATE(p)                              ! eliberare pointer p
 END FUNCTION realoca
END MODULE realocare_m

PROGRAM test_realocare
USE realocare_m              ! specificarea utilizarii modulului
 IMPLICIT NONE               ! declarație obisnuita in Fortran 90
 REAL, POINTER, DIMENSION(:) :: p
 INTEGER :: j, n_elem = 2
 ALLOCATE(p(1:n_elem))
 p(1) = 12345
 p => realoca(p, 10000)    ! observati maniera specifica de asignare
 WRITE(*,*) "sunt alocate ", n_elem, size(p), " elemente"
 WRITE(*,*) "p(1)=", p(1)
END PROGRAM test_realocare
```

Pointerii se pot folosi şi la alcătuirea unor structuri de date dinamice complexe, cum ar fi listele înlănțuite, arborii binari etc. Acest lucru este posibil deoarece o variabilă de tip derivat poate conține pointeri care să indice către sine sau către alte obiecte similare (cu condiția ca țintele să fie valide). În exemplul următor prezentăm o posibilă implementare a unei cozi:

```
PROGRAM coada
 IMPLICIT NONE
 TYPE :: tip_element          ! specificarea unei structuri de date
  CHARACTER(20) :: sir
                              ! pointer catre un obiect similar:
  TYPE(tip_element), POINTER :: urmator
 END TYPE tip_element
 TYPE(tip_element), POINTER :: fata, spate, pozitie
 CHARACTER(20) :: tampon
 INTEGER :: stare
 NULLIFY(fata, spate)         ! setarea initial goala a cozii, apoi
                              ! ciclu pentru citirea unor caractere
 DO                           ! tastate de utilizator, in tampon
  READ(*, "(A)", IOSTAT=stare) tampon
  IF(stare /= 0) EXIT         ! iesirea din ciclu
  IF( .NOT. ASSOCIATED(fata)) THEN
   ALLOCATE(fata)             ! crearea locatiei pentru prima pozitie
   spate => fata              ! spre care indica fata si spate
  ELSE
   ALLOCATE(spate%urmator)    ! locatia pentru pozitia urmatoare
   spate => spate%urmator     ! spre care indica spate
  END IF
  spate%sir = tampon          ! stocarea sir-ului in pozitia noua si
  NULLIFY(spate%urmator)      ! marcarea acesteia ca ultima din coada
 END DO
              ! dupa iesirea din ciclu urmeaza parcurgerea cozii
              ! cu afisarea continutului fiecarui element:
 pozitie => fata              ! repozitionare pe inceputul cozii,
                              ! si ciclu pentru
 DO WHILE(ASSOCIATED(pozitie))
  WRITE(*,*) pozitie%sir      ! afisarea continutului din pozitia
                              ! curenta, si
  pozitie => pozitie%urmator  ! asignarea pointerului la urmatorea
                              ! pozitie din coada
 END DO
 STOP
END PROGRAM coada
```

4.11 ALTE ASPECTE

Valorile de tip binar, octal, şi hexadecimal se pot citi sau scrie utilizând descriptorii noi de tipul B, O, şi Z, iar constantele numerice de acest fel se scriu cu litera corespunzătoare tipului în fața valorii citate (de exemplu, valoarea decimală `15` se poate scrie sub formă binară ca: `B"1111"`, sub formă octală ca: `O"17"`, sub formă hexadecimală ca: `Z"F"`). Specificațiile DATA pot să conțină de asemenea constante binare, octale şi hexadecimale.

4.11.1 Funcții intrinseci şi facilități noi

Standardul limbajului Fortran a fost extins şi cu o serie de funcții intrinseci ce facilitează manipulările valorilor la nivel de biți. Numerotarea biților se face de regulă dinspre dreapta spre stânga, pornind de la poziția zero (sau de la capătul cel mai puțin semnificativ).

Tabel cu noile funcții intrinseci pentru operații la nivel de bit asupra valorilor întregi:

Funcție	Efect
`BIT_SIZE(t)`	Returnează numărul biților din variabila de tipul *t*.
`BTEST(i, p)`	Testează bitul de pe poziția *p* din valoarea *i* de tip întreg.
`IAND(i, j)`	ŞI logic între două argumente de tip întreg.
`IBCLR(i, p)`	Goleşte conținutul bitului de pe poziția *p* din întregul *i* (valoarea bitului va fi zero).
`IBCHNG(i, p)`	Inversează valoarea bitului de pe poziția *p* din întregul *i*.
`IBITS(i, p, l)`	Extrage şirul de biți cu lungimea *l* începând de la *p* din *i*.
`IBSET(i, p)`	Setează valoarea bitului de pe poziția *p* din întregul *i* pe unu.
`IEOR(i, j)`	SAU exclusiv între două argumente de tip întreg.
`IOR(i, j)`	SAU inclusiv între două argumente de tip întreg.
`ISHFT(i, n)`	Translatare logică de biți la stânga (sau la dreapta dacă *n* este negativ) cu *n* poziții în cadrul lui *i*.
`ISHFTC(i, n)`	Deplasare circulară logică de biți la stânga (sau la dreapta dacă *n* este negativ) cu *n* poziții în cadrul lui *i*.
`NOT(i)`	Complementul logic al argumentului *i* de tip întreg.

Noile funcții intrinseci FLOOR şi MODULO au sintaxe şi efecte similare cu cele ale funcțiilor AINT şi MOD deja cunoscute, cu deosebiri doar în cazul numerelor negative. Funcția CEILING, similară acestora ca formă, rotunjeşte argumentul în sus, la cel mai apropiat întreg.

Funcția intrinsecă TRANSFER poate fi folosită pentru copierea biților dintr-o valoare de un anume tip într-o valoare de alt tip (o alternativă mai sigură decât artificiile posibile prin declarații EQUIVALENCE). Iată spre exemplu, cum se poate folosi această funcție pentru testarea unui calculator din cadrul unui program, cu privire la reprezentarea caracterelor:

```
LOGICAL, PARAMETER :: bigend = IACHAR(TRANSFER(1,"a")) == 0
```

Dacă platforma este de tip *big-endian* atunci parametrul *bigend* va primi valoarea `.TRUE.` (altfel va avea valoarea `.FALSE.`).

Dintre funcțiile numerice intrinseci menționăm funcțiile TINY (care returnează cel mai mic număr pozitiv de tipul şi felul argumentului) şi HUGE (care returnează cel mai mare număr pozitiv de tipul şi felul argumentului), care acceptă ca argument atât valori întregi cât şi valori reale, de orice fel. Deşi există încă multe alte funcții intrinseci noi, ne limităm doar la a menționa câteva: BIT_SIZE, DIGITS, EPSILON, MAXEXPONENT, MINEXPONENT, PRECISION, RADIX, RANGE etc.

4.11.2 Subprograme predefinite

Pe lângă multitudinea funcțiilor noi există şi câteva subprograme interne, apelabile din orice unitate de program. Dintre acestea menționăm:

`DATE_AND_TIME([data][,timp][,zona][,valori])` – returnează data şi ora curentă sub forma unui şir sau sub forma unui tablou de valori întregi;

`RANDOM_NUMBER([HARVEST=]tablou)` – returnează un tablou de valori pseudoaleatoare uniform distribuite în intervalul 0—1;

`RANDOM_SEED([măsuma][,pune][,ia])` – inițializează sau recuperează valoarea inițială a generatorului de numere pseudoaleatoare;

`SYSTEM_CLOCK([contor][,rata][,maxim])` – permite tratarea unor intervale de timp în funcție de ceasul calculatorului (în Fortran 95 există o rutină mai curată: CPU_TIME).

4.11.3 Aspecte legate de evoluția limbajului

Anumite aspecte din Fortran au fost considerate atât de depăşite încât au fost excluse din standardele noi ale limbajului. După saltul imens produs de apariția standardului Fortran 90 limbajul a fost dezvoltat în continuare, iar până în 1996 s-a elaborat o nouă versiune sub specificația de Fortran 95. Diferențele dintre aceste două versiuni nu sunt atât de mari, însă caracteristicile considerate depăşite au fost lăsate uitării. Folosirea lor nu este recomandată, chiar dacă unele compilatoare Fortran 95 semnalează doar erori de atenționare la întâlnirea acestora. Iată câteva dintre aceste aspecte problematice:

- Cicluri DO cu variabilă de control de tip real (sau dublă precizie).
- Cicluri DO finalizate prin alte instrucțiuni decât CONTINUE sau END DO.
- Două sau mai multe cicluri DO (incluse) finalizate prin aceeaşi instrucțiune.
- Instrucțiunea IF aritmetică.
- Folosirea constantelor Hollerith (sau descriptorului H) în listele de descriptori din instrucțiunile FORMAT.
- Folosirea instrucțiunii PAUSE (se poate înlocui lesne cu o instrucțiune de citire fără listă de intrare).
- Folosirea instrucțiunii ASSIGN (şi implicit a instrucțiunii GO TO asignate, a etichetelor asignate pentru FORMAT etc.).
- Folosirea facilității de revenire (RETURN) alternativă.
- Specificarea unei ramuri către instrucțiunea END IF din afara blocului IF (posibil în Fortran 77 din greşeală).

Excluderea acestor facilităţi nu reprezintă o problemă, cu atât mai puţin pentru cei ce nu uzau de ele. Există şi alte aspecte care deşi sunt utilizate în mod obişnuit în Fortran 77, sunt considerate ca fiind redundante şi ca atare se recomandă a fi evitate în scrierea surselor. De exemplu: formatul fix, tipurile de date implicite (nedeclararea explicită a tipurilor), blocurile comune (COMMON), tablourile cu mărime presupusă, instrucţiunile INCLUDE, EQUIVALENCE, ENTRY, unităţile de program BLOCK DATA etc.

Dintre principalele caracteristici noi ale limbajului Fortran 95 menţionăm:

- instrucţiunea FORALL (şi facilităţile noi oferite şi pentru WHERE);
- subprogramele pure (PURE) şi elementare (ELEMENTAL) definite de utilizator;
- starea de asociere iniţială a pointerilor, prin `=> NULL()`;
- iniţializarea implicită a obiectelor de tip derivat;
- referirile la proceduri pure în expresiile de specificare;
- dealocarea automată a tablourilor alocabile;
- formatarea prin specificator cu lungime zero produce considerarea numărului minim de poziţii necesare (de exemplu: `I0`);
- noua funcţie intrinsecă CPU_TIME;
- schimbări ale unor funcţii intrinseci etc.

Pentru detalii suplimentare vă recomandăm consultarea sitului realizat de Bo Einarsson (la adresa http://www.nsc.liu.se/~boein/f77to90/f95.html).

Ultima variantă dezvoltată a fost botezată Fortran 2000. Dintre noutăţile aduse de aceasta şi facilităţile oferite menţionăm:

- performanţe ridicate în domeniul calculelor ştiinţifice şi inginereşti (operaţii asincrone de citire/scriere, tratarea excepţiilor de virgulă flotantă, interval aritmetic);
- abstractizarea datelor cu extensibilitatea oferită către utilizatori (componente alocabile, intrări/ieşiri de tip derivat);
- orientare pe obiecte (constructori/destructori, moştenire, polimorfism);
- tipuri derivate parametrizate;
- pointeri la proceduri;
- internaţionalizare;
- interoperabilitate cu limbajul C.

Ca şi resurse disponibile (standarde şi versiuni ale limbajului, compilatoare, medii de programare, generatoare de interfeţe, programe de conversie, programe gratuite, produse comerciale, documentaţii, liste cu întrebări frecvente şi forumuri etc.) vă recomandăm consultarea adresei http://www.fortran.com/fortran/ .

CAPITOLUL 5: G77

5.1 COMPILATORUL GNU FORTRAN 77

GNU Fortran 77 (cunoscut şi ca *g77*) este un compilator realizat de către o echipă condusă de Craig Burley în cadrul organizației *Free Software Foundation* (Fundația pentru Software Liber) cu scopul de a sprijini dezvoltarea programelor în acest limbaj. Oferă suport complet pentru fişiere sursă scrise în Fortran 77, acceptând şi unele extensi comune derivate în specificații de tip Fortran 90. Fişierele executabile (programele) rezultate sunt la fel de rapide ca cele realizate cu compilatoare comerciale. Compilatorul g77 este accesibil prin internet tuturor celor interesați pornind de la următoarea adresă, ca parte a pachetului *gcc* (*GNU Compiler Collection*):

http://www.gnu.org/directory/gcc.html
sau
http://www.gnu.org/software/gcc/gcc.html

Pentru cei ce folosesc sistemul de operare Windows, merită să menționăm şi unul dintre siturile educaționale, cum ar fi cea de la Universitatea Statului Utah, la adresa:

http://www.engineering.usu.edu/cee/faculty/gurro/Classes/Classes_Main.htm

unde pe lângă pachetul G77.ZIP există şi cursuri de inițiere în programare cu ajutorul limbajului Fortran 77, folosind şi facilitățile noi permise de compilatorul g77.

Sursele programelor pot fi redactate cu ajutorul oricărui editor de text ce poate salva fişiere text curate (ASCII). Compilatorul g77 lansat sub sistemul de operare Windows rulează în mod normal sub fereastră DOS. Acest aspect implică respectarea convențiilor din DOS pentru specificatorii de fişiere şi directoare (cel mult 8 caractere alfanumerice pentru nume şi 3 pentru extensie). Există însă şi câteva interfețe grafice accesibile (cu licență de utilizare liberă) care conțin şi editoare de text adecvate pe lângă compilatorul g77, asemănătoare cu medii de dezvoltare comerciale, uneori prea scumpe. Dintre cele mai simple asemenea interfețe amintim două: Vfort (printre primele apărute, autorii fiind N. şi P. Vabişcevici din Rusia, http://www.imamod.ru/~vab/vfort/), şi Force2 (un proiect realizat şi dezvoltat de către G. Lepsch Guedes, http://www.forceproject.hpg.com.br/).

Pentru a creea fişiere executabile cu ajutorul compilatorului g77, se poate folosi o linie de comandă de genul:

```
G77 nume1.F[ –op][ nume2.EXE]
```

unde: `G77` – numele compilatorului GNU Fortran 77;
`nume1.F` – specificatorul fişierului sursă;
`–op` – opțiune pentru compilare;
`nume2.EXE` – specificatorul fişierului executabil ce se creează.

Fişierul executabil *nume2.exe* rezultat poate fi lansat în execuție prin invocarea numelui *nume2* într-o linie nouă de comandă, cu condiția ca să nu existe erori care să impieteze asupra generării lui.

Dacă în timpul compilării sursei erorile de sintaxă afişate în fereastra DOS curentă sunt prea numeroase, depăşind zona afişajului curent, se poate redirecta ieşirea *stderr* (*Standard Error*) într-un fişier text. Această facilitate este oferită de cele mai multe medii de programare, dar poate fi realizată şi prin programul ETIME.EXE (program de temporizare cu redirectare cuprins în pachetul G77.ZIP oferit de Gilberto E. Urroz de la Utah State University, la adresa deja menționată) sub DOS printr-o linie de comandă de forma:

```
ETIME –2specfis G77 nume1.F
```

unde: `ETIME` – numele programului de redirectare;
`–2` – opțiune pentru redirectarea descriptorului de fişiere 2 (*stderr*);
`specfis` – specificatorul fişierului în care se vor înregistra mesajele redirectate;
`G77` – numele compilatorului GNU Fortran 77 (*g77.exe*);
`nume1.F` – specificatorul fişierului sursă.

În acest caz nu se va genera nici un mesaj (ecou) pe ecran, nu se va crea nici imagine obiect nici imagine executabilă pentru fişierul sursă *nume1.F*, însă fişierul specificat prin *specfis* va conține toate mesajele ce ar fi apărut pe ecran în urma compilării cu g77. Vizualizând conținutul fişierului *specfis* generat se poate depana mai uşor sursa conținută în *nume1.F*.

5.2 COMPILAREA CU G77

Comanda G77 sub forma primului exemplu din subcapitolul precedent va determina compilarea şi editarea legăturilor din programul scris, generând un fişier executabil ce va putea fi rulat sub DOS sau Windows 9x/NT (în fereastră DOS). Dacă numele fişierului executabil nu se specifică în linia de comandă, se va genera un fişier cu numele identic cu cel al sursei (dar cu extensia .EXE). Există mai multe opțiuni ce se pot specifica după numele fişierului sursă (atenție la litere mici şi la majuscule!), dintre care menționăm:

`–c`	doar compilare (*Compile-only*), se va genera doar imagine obiect fără imagine executabilă (rezultă doar fişier cu extensia .OBJ);
`–ffree–form`	pentru fişier sursă redactat sub formă liberă;
`–fpedantic`	va avertiza asupra codului neportabil sau nestandard;
`–fno–automatic`	pentru alocare statică la toate variabilele (similar cu SAVE universal):
`–fno–backslash`	va interpreta \ ca şi caracter normal în şiruri;
`–fvxt`	pentru interpretarea anumitor sintaxe de tipul VAX Fortran;

`–g`	va produce informații pentru depanare;
`–Idirector`	pentru specificarea directorului cu fişiere de inclus (prin INCLUDE);
`–O`	pentru optimizarea codului generat;
`–Wimplicit`	va avertiza asupra numelor de date cu tip neexplicit;
`–Wuninitialised`	va avertiza în anumite cazuri asupra variabilelor fără valoare (dacă s-a folosit opțiunea `–O`);
`–Wall`	va genera mesaje de avertizare referitoare la variabile neutilizate sau nedeclarate (ca cele două opțiuni precedente combinate).

În linia de comandă se pot specifica mai multe fişiere sursă, se admite şi caracterul ***** în specificații (conform convențiilor din DOS). De asemenea, se pot specifica fişiere compilate (imagini obiect cu extensia .OBJ) şi fişiere de bibliotecă (numite *arhivă* în jargonul utilizatorilor de Unix, fişiere cu extensia .A). La specificarea opțiunilor trebuie să avem grijă la respectarea formei de scriere, însă specificatorii de fişiere sub DOS (şi sub Windows) nu sunt sensibili la mărimea caracterelor utilizate (caracterele mici sunt echivalente cu majusculele corespunzătoare). Iată şi un scurt exemplu pentru ilustrarea compilării cu G77:

Creeați un fişier text (ASCII) cu un editor convenabil (de exemplu NotePad sub Windows), cu următorul conținut, ținând cont de faptul că în format fix instrucțiunile încep din coloana a 7-a şi se termină până în coloana 72:

```
C2345678901234567890123
      Program TEST
      integer i
      real x(10)
      do 1 i=1,10
       x(i)=i*i
1     write(*,*)i,x(i)        ! ciclu finalizat fara CONTINUE
      end
```

Salvați fişierul sub numele TEST.F alături de programul G77.EXE şi deschideți o fereastră DOS (având grijă să vă aflați în directorul în care aveți compilatorul şi fişierul sursă creat). Pentru a compila sursa scrisă şi pentru a creea o imagine executabilă tastați linia de comandă:

```
G77 TEST.F –O TEST
```

Dacă ați lucrat corect, după terminarea procesului lansat se va afişa din nou prompterul curent din fereastra DOS. În cazul în care vi se afişează mesaje de eroare, citiți-le cu atenție, dacă trebuie editați din nou fişierul sursă şi corectați inadvertențele înainte de a-l salva din nou. Pentru a verifica existența programului generat puteți tasta comanda:

```
DIR TEST.*
```

Dacă pe ecran vi se afişează printre altele şi specificatorul de fişier TEST.EXE, atunci puteți lansa în execuție programul creat, tastându-i numele:

```
TEST
```

Programul va afişa pe ecran valorile de la 1 la 10 cu pătratul lor, succesiv. Prin comanda DIR puteți vedea şi diferența de mărime dintre fişierul sursă (TEST.F) şi programul creat (TEST.EXE). Evident, veți putea creea fişiere sursă şi programe şi în alte directoare, pentru acest fapt însă veți fi nevoiți să vă configurați mediul de lucru corespunzător (setarea variabilei PATH etc.).

Dacă folosiți mediile de programare cu interfață grafică VFort sau Force2 amintite, atunci va trebui să țineți cont de setările şi opțiunile acestora, compilarea şi editarea legăturilor realizându-se cu comenzile grafice corespunzătoare (prin activarea butoanelor oferite prin interfața grafică). Pentru informații mai detaliate recomandăm consultarea adreselor de web menționate.

5.3 BIBLIOTECI PENTRU G77

Fortran fiind un limbaj de programare dezvoltat în scopuri ştiințifice, există o varietate foarte mare de biblioteci matematice, mai ales cu implementări de metode numerice. Ne rezumăm însă să amintim doar două pachete disponibile prin internet (pornind de la adresa:

http://www.geocities.com/Athens/Olympus/5564/ ,

la care se găseşte pagina dedicată compilatorului G77 pentru variantele pe 32 de biți ale sistemului de operare Windows) şi compatibile cu GNU Fortran 77.

Primul pachet este biblioteca matematică SLATEC, dezvoltată la Laboratoarele Naționale Americane (din Los Alamos, Lawrence Livermore, NIST, Oak Ridge, Sandia etc.), ce conține 902 de module apelabile de către utilizatori, fiind de fapt o colecție compusă din rutinele considerate ca cele mai utile din cadrul altor pachete de bibliotecă (BLAS, LINPACK, EISPACK, SLAP, FFTPACK, FISHPACK, LLSQ, MINPACK, MP, PCHIP, QUADPACK şi SPLPACK).

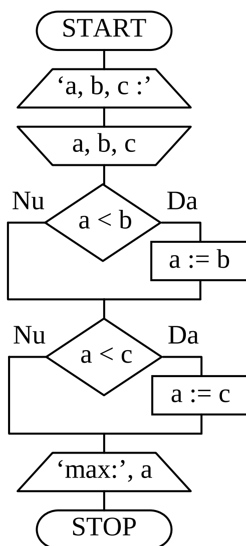
Al doilea pachet este o bibliotecă grafică cu numele PSPLOT, ce permite generarea imaginilor în format PS (*PostScript*). Acest format este independent de sistemul de operare folosit, fiind acceptat şi utilizat de către multe periferice de imprimare, pentru vizualizarea imaginilor pe ecran fiind însă nevoie de o aplicație corespunzătoare (cum ar fi pachetele Aladdin Ghostscript şi GSView, disponibile sub licențe libere pentru diverse platforme).

CAPITOLUL 6: EXERCIȚII

6.1 EXERCIȚII ELEMENTARE INTRODUCTIVE

6.1.1 Să se schițeze o schemă logică pentru alegerea celei mai mari valori dintre *a*, *b*, şi *c*, şi să se scrie un program FORTRAN pe baza schemei respective.

Soluție propusă:




```
!2345678901234567890123456789012345678
!randul de mai sus arata doar coloana.
! incepem direct cu
! afisarea mesajului pentru cerere:
      write(*,*)' a, b, c: '

! citirea valorilor pentru a, b, c:
      read(*,*)a,b,c

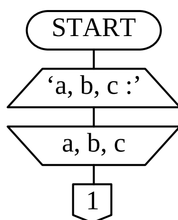
! testarea primei conditii
! (si in caz afirmativ atribuire):
      if(a.lt.b) a=b


! testarea urmatoarei conditii
! (si in caz afirmativ atribuire):
      if(a.lt.c) a=c


! afisarea rezultatului stocat in a:
      write(*,*)' max: ', a
      stop
      end
```

6.1.2 Să se schițeze o schemă logică pentru ordonarea crescătoare a valorilor *a*, *b*, şi *c*, după care să se scrie un program FORTRAN pe baza schemei întocmite.

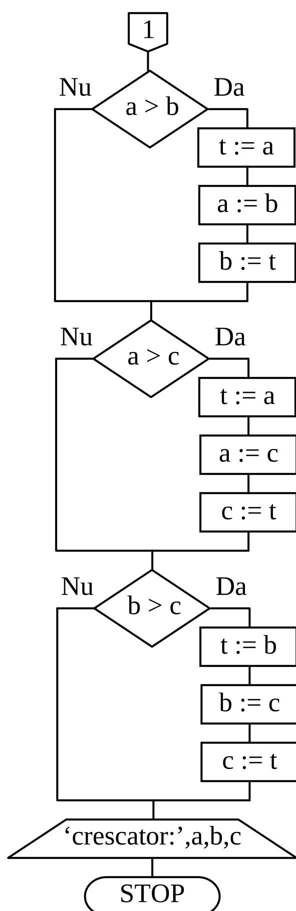
Soluție propusă:




```
! incepem cu declaratia de program:
      program abcordc

! afisarea mesajului pentru cerere:
      write(*,*)' a, b, c: '
! citirea valorilor pentru a, b, c:
      read(*,*)a,b,c

! schema logica este continuata pe
! pagina urmatoare, la fel si
! programul, scris alaturat.
```




```
! continuarea schemei
! si a programului,

! cu testarea primei conditii
! (in caz afirmativ schimbam valorile
! dintre a si b cu ajutorul lui t):
      if(a.gt.b) then
       t=a
       a=b
       b=t
      endif

! testarea urmatoarei conditii
! (in caz afirmativ schimbam valorile
! dintre a si c cu ajutorul lui t):
      if(a.gt.c) then
       t=a
       a=c
       c=t
      endif

! testarea urmatoarei conditii
! (in caz afirmativ schimbam valorile
! dintre b si c cu ajutorul lui t):
      if(b.gt.c) then
       t=b
       b=c
       c=t
      endif

! afisarea rezultatului:
      write(*,*)' crescator: ',a,b,c
! si implicit sfarsitul programului
      end
```

6.1.3 Să se scrie un program FORTRAN care în urma citirii de la tastatură a unui număr întreg din intervalul [1,7], afişează ziua din săptămână care corespunde acelui număr (1 ≡ luni, 2 ≡ marți etc.).

Soluție propusă:

```
      program saptamana
! --- programul indica ziua saptamanii pe baza unui nr. intreg
!     din intervalul [1,7]
!     nu a fost declarat explicit tipul variabilelor, in mod implicit
!     variabilele ale caror nume incepe cu i, j, k, l, m, n sunt de
!     tip intreg, iar restul de tip real (de evitat acest stil!)
33    write(*,*)' dati un nr. intreg din intervalul [1,7] : '
      read(*,*)i
      write(*,*)             ! se tipareste o linie goala
```

```
!     testare caz
      select case (i)
      case (1)
       write(*,*) 'ziua corespunzatoare nr. ales este LUNI'
      case (2)
       write(*,*) 'ziua corespunzatoare nr. ales este MARTI'
      case (3)
       write(*,*) 'ziua corespunzatoare nr. ales este MIERCURI'
      case (4)
       write(*,*) 'ziua corespunzatoare nr. ales este JOI'
      case (5)
       write(*,*) 'ziua corespunzatoare nr. ales este VINERI'
      case (6)
       write(*,*) 'ziua corespunzatoare nr. ales este SAMBATA'
      case (7)
       write(*,*) 'ziua corespunzatoare nr. ales este DUMINICA'
      case default
           write(*,*) 'nr. eronat !! '
           write(*,*)
           goto 33
      end select
      end
```

6.1.4 Să se scrie un program FORTRAN care pe baza lungimilor a 3 segmente, decide dacă acestea pot forma un triunghi.

Soluție propusă:

```
      program triunghi
! --- decide daca 3 segmente pot fi laturile unui triunghi
!     pe baza inegalitatilor cunoscute de la geometrie
!
!     se foloseste regula implicita
!     deci fara declaratii explicite ale tipului variabilelor
!
      write(*,*)' dati lungimea primului segment : '
      read(*,*)s1
      write(*,*)' dati lungimea celui de-al doilea segment : '
      read(*,*)s2
      write(*,*)' dati lungimea celui de-al treilea segment : '
      read(*,*)s3
!     testare inegalitati
      if((s1.lt.(s2+s3)).and.(s2.lt.(s1+s3)).and.
     +   (s3.lt.(s1+s2))) then
        write(*,*) 'segmentele POT fi laturile unui triunghi'
      else
        write(*,*) 'segmentele NU pot fi laturile unui triunghi'
      endif
      end
```

6.1.5 Să se scrie un program FORTRAN care afişează pe ecran următorul tabel, folosind instrucțiunea FORMAT.

```
*********************
|  N  |   Npatrat   |
*********************
|  1  |      1      |
---------------------
|  2  |      4      |
---------------------
|  3  |      9      |
---------------------
|  4  |     16      |
---------------------
|  5  |     25      |
---------------------
|  6  |     36      |
---------------------
|  7  |     49      |
---------------------
|  8  |     64      |
---------------------
|  9  |     81      |
---------------------
| 10  |    100      |
---------------------
```

Soluție propusă:

```
      program tabel
! --- creeaza un tabel, utilizand FORMAT
!
      implicit none
      integer n
100   format(1x,a21)
      print 100,  '*********************'
      print*,     '|  N  |   Npatrat   |'
      print 100,  '*********************'
      do n=1,10
        print 200,'| ',n,'  |     ',n*n,'  |'
        print*,   '---------------------'
      enddo
200   format(1x,a2,i2,a8,i3,a6)
      end
```

6.2 EXERCIȚII CU EXPRESII ARITMETICE

6.2.1 Să se scrie un program FORTRAN care calculează valorile expresiei $e = \frac{a+2b}{c-d}$ dacă *a*, *b*, *c*, şi *d* iau toate valorile întregi din intervalul [0,2].

Soluție propusă:

```
      program expresie
! --- calculul valorii unei expresii in functie
!     de 4 variabile care parcurg un interval de nr. intregi
      implicit none
      integer a,b,c,d
      integer i              ! contor al nr. de operatii/linii
      real e
!     initializare contor
      i=0
!     4 cicluri DO imbricate
      do 1 a=0,2
       do 2 b=0,2
        do 3 c=0,2
         do 4 d=0,2
          i=i+1
! --- se pune problema afisarii unui nr. dorit de linii pe ecran
!     adica derulare controlata a rezultatelor
! --- sa presupunem ca se doreste afisarea a cate 21 de linii
          if(mod(i,21).eq.0) pause
! --- functia MOD testeaza daca i este multiplu de 21
! --- instructiunea PAUSE opreste temporar executia programului
! --- urmeaza acum un test de numitor nul
          if(c.eq.d) then
           write(*,*) 'a=',a,' b=',b,' c=',c,' d=',d,
     &                '    e nu se poate calcula'
          else
           e=(a+2.0*b)/(c-d)
           write(*,*) 'a=',a,' b=',b,' c=',c,' d=',d,'    -> e=',e
          endif
4        continue
3       continue
2      continue
1     continue
! --- cite operatii s-au efectuat
      write(*,'(//,1x,a14,i2,a9)')'s-au efectuat ',i,' operatii'
! --- pt. afisare au fost utilizati descriptori de format
      end
```

6.2.2 Să se scrie un program FORTRAN care calculează suma $1^2 + 2^2 + 3^2 + ... + 10^2$.

Soluție propusă:

```
      program spatrat
! --- calculul sumei patratelor primelor 10 nr. naturale
```

```
      implicit none
      integer n
      parameter (n=10)      ! in F90, s-ar fi scris INTEGER,PARAMETER::n=10
      integer s,i
      s=0
      do i=1,n
        s=s+i*i
      enddo
      write(*,*) 'suma patratelor primelor ',n,' nr. naturale este : ',
     &            s
      end
```

6.2.3 Să se scrie un program FORTRAN care calculează media aritmetică a primelor 10 numere naturale impare. Se cere deci calculul expresiei:

$$\frac{1+3+5+7+9+11+13+15+17+19}{10}$$

Soluție propusă:

```
      program medimpar
! --- calculul mediei aritmetice a primelor 10 nr. naturale impare
!
      implicit none
      integer n
      parameter (n=10)
      integer s,i
      integer k !contor
      s=0 !initializarea sumei
      i=1 !initializarea primului nr. impar
      k=1 !initializarea contorului
333   s=s+i
      i=i+2
      k=k+1
      if(k.le.n) goto 333
      write(*,*) 'media aritmetica a primelor ',n,
     &           ' nr. naturale impare este : ',s*1./n
      end
```

6.2.4 Să se scrie un program FORTRAN care decide dacă un triunghi este echilateral, pe baza coordonatelor vârfurilor.

Soluție propusă:

```
      program echilat
! --- decide daca un triunghi este echilateral
!     pe baza coordonatelor varfurilor
!
      implicit none
      real x1,y1,x2,y2,x3,y3
      real l1,l2,l3
```

```
      print*,'dati coordonatele varfurilor triunghiului :'
      write(*,'(/,a3,$)')'x1='
      read*,x1
      write(*,'(a3,$)')'y1='
      read*,y1
      write(*,'(/,a3,$)')'x2='
      read*,x2
      write(*,'(a3,$)')'y2='
      read*,y2
      write(*,'(/,a3,$)')'x3='
      read*,x3
      write(*,'(a3,$)')'y3='
      read*,y3
      l1=sqrt((x2-x1)**2+(y2-y1)**2)
      l2=sqrt((x2-x3)**2+(y2-y3)**2)
      l3=sqrt((x3-x1)**2+(y3-y1)**2)
! urmeaza un exemplu de folosire periculoasa a operatorului .EQ.
! in realitate, nr. reale in calculator pot fi "egale" doar in
! limitele unei tolerante admise
      if(l1.eq.l2.and.l2.eq.l3) then
        print*
        print*,'triunghiul ESTE echilateral'
      else
        print*
        print*,'triunghiul NU ESTE echilateral'
      endif
      end
```

6.2.5 Să se scrie un program FORTRAN care calculează valorile funcției:

$$f(x) = x^2 + \sin(x)$$

în intervalul [0,2], cu pasul 0,1.

Soluție propusă:

```
      program functie
! --- calculeaza valorile functiei x**2+sin(x) in [0,2]
!     cu pasul 0.1
!
      implicit none
      real x,y
      integer i
      do i=0,20,1
! este recomandabil a utiliza contor de tip intreg in ciclul DO
! chiar daca F90 accepta si contor nr. real, exista pericolul
! de a nu "inchide" ciclul DO
        x=i/10.
        y=x*x+sin(x)
        write(*,'(a9,f3.1,a7,f8.6)')'pentru x=',x,'  f(x)=',y
! mai simplu, se putea scrie neformatat           PRINT*,x,y
      enddo
      end
```

6.2.6 Să se scrie un program FORTRAN care rezolvă un sistem liniar de 2 ecuații cu 2 necunoscute.

Soluție propusă:

```
      program sistem
! --- rezolva un sistem liniar 2x2, de tipul
!             ax+by=c
!             dx+ey=f
!
      implicit none
      real a,b,c,d,e,f
      real delta,x,y
      print*,'dati a,b,c din prima ecuatie : ax+by=c '
      read*,a,b,c
      print*,'dati d,e,f din a doua ecuatie : dx+ey=f '
      read*,d,e,f
      delta=a*e-b*d
      if(delta.eq.0) then
        if(b*f.eq.c*e) then
          print*
          print*,'sistemul este compatibil nedeterminat'
        else
          print*
          print*,'sistemul este incompatibil'
        endif
      else
        x=(c*e-b*f)/delta
        y=(a*f-c*d)/delta
        print*
        print*,'sistemul este compatibil determinat'
        print*,'x=',x
        print*,'y=',y
      endif
      end
```

6.3 EXERCIȚII CU TABLOURI DE DATE

6.3.1 Să se scrie un program FORTRAN care determină câte numere pare există într-un şir de *n* numere naturale. Seria de *n* numere naturale va fi introdusă de la tastatură.

Soluție propusă:

```
      program par
! --- determina cate nr. pare exista intr-un sir de N nr. naturale
!
      implicit none
      integer a(50)        ! dimensiunea maxima a sirului este 50
      integer n            ! dimensiunea efectiva a sirului va fi n
      integer i,k
503   print*,'dati nr. de elemente din sir : N < 50'
```

```
      read*,n
! --- verificare
      if(n.lt.1.or.n.gt.50) goto 503
! --- introducerea elem. sirului
      print*
      print*,'Introduceti elem. sirului:'
      print*
      do i=1,n
        write(*,'(1x,a2,i2,a2,$)') 'a(',i,')='
        read(*,*) a(i)
      enddo
! --- initializare contor
      k=0
! --- determinarea nr. de elemente pare
      do i=1,n
       if(mod(a(i),2)==0) k=k+1
      enddo
! --- afisarea rezultatului
      print*
      write(*,'(a26,i2,a9)')'In sirul introdus exista ',k,' nr. PARE'
      print*
      end
```

6.3.2 Să se scrie un program FORTRAN care determină intersecția a două mulțimi având fiecare *m* elemente numere întregi.

Soluție propusă:

```
      program intersectie
! --- intersectia a 2 multimi A,B avand ambele M elemente intregi
!
      implicit none
      integer a(50),b(50),inters(50)    ! dimensiunea maxima a multimilor
      integer m                       ! dimensiunea efectiva a multimilor
      integer i,j,k
999   print*,'dati dimensiunea efectiva a multimilor : M < 50'
      read*,m
! --- sa ne asiguram ca utilizatorul respecta cerinta :
      if(m.lt.1.or.m.gt.50) goto 999
! --- introducerea elem. multimii A
      print*
      print*,'Elem. multimii A (nr. intregi distincte, max. 3 cifre!):'
      print*
      do i=1,m
22      write(*,'(1x,a2,i2,a2,$)') 'a(',i,')='
        read(*,*) a(i)
! --- verificarea prezentei elem. duble
        do j=1,i-1
         if(a(i).eq.a(j)) then
           ! exista elem. duble
           print*,' *** Elementele trebuie sa fie toate distincte ***'
           print*
           goto 22
         endif
```

```
          enddo
        enddo
! --- introducerea elem. multimii B
        print*
        print*,'Elem. multimii B (nr. intregi distincte, max. 3 cifre!):'
        print*
        do i=1,m
33        write(*,'(1x,a2,i2,a2,$)') 'b(',i,')='
          read(*,*) b(i)
! --- verificarea prezentei elem. duble
          do j=1,i-1
           if(b(i).eq.b(j)) then
             ! exista elem. duble
             print*,' *** Elementele trebuie sa fie toate distincte ***'
             print*
             goto 33
           endif
          enddo
        enddo
! --- initializare contor elemente comune
        k=0
! --- determinarea intersectiei
        do i=1,m
         do j=1,m
          if(a(i).eq.b(j)) then
           k=k+1
           inters(k)=a(i)
          endif
         enddo
        enddo
! --- afisarea rezultatului
        print*
        print*,'Intersectia multimilor este :'
        print*
        if(k.ne.0) then
         do i=1,k
          write(*,'(i4,$)') inters(i)
         enddo
        else
         print*,' vida'
        endif
        end
```

6.3.3 Să se scrie un program FORTRAN care efectuează produsul a două matrici de numere reale.

Soluție propusă:

```
        program produs
! --- produsul a 2 matrici reale AxB
*       A este de tip MxN
*       B este de tip NxP
!
        implicit none
```

```
      real a(50,40),b(40,60),prod(50,60)    ! dimens. maxime ale matricilor
      integer m,n,n2,p                      ! dimensiunile efective
      integer i,j,k
1     print*,'dati dimensiunile efective ale matricii A : M<50,N<40'
      read*,m,n
! --- verificare dimensiuni A
      if(m.lt.1.or.m.gt.50.or.n.lt.1.or.n.gt.40) goto 1
2     print*,'dati dimensiunile efective ale matricii B : N<40,P<60'
      read*,n2,p
! --- verificare compatibilitate
      if(n.ne.n2) then
       print*,'Inmultirea nu poate fi facuta !'
       goto 1
      endif
! --- verificare dimensiuni B
      if(p.lt.1.or.p.gt.60) goto 2
! --- introducerea elem. matricii A
      print*
      print*,'Elementele matricii A :'
      print*
      do i=1,m
       do j=1,n
        write(*,'(1x,a2,i1,a1,i1,a2,$)') 'a(',i,',',j,')='
        read(*,*) a(i,j)
       enddo
      enddo
! --- introducerea elem. matricii B
      print*
      print*,'Elementele matricii B :'
      print*
      do i=1,n
       do j=1,p
        write(*,'(1x,a2,i1,a1,i1,a2,$)') 'b(',i,',',j,')='
        read(*,*) b(i,j)
       enddo
      enddo
! --- inmultirea propriu-zisa
      do 44 i=1,m
       do 55 j=1,p
        prod(i,j)=0.
        do 66 k=1,n
         prod(i,j)=prod(i,j)+a(i,k)*b(k,j)
   66   continue
   55  continue
   44 continue
! --- afisarea rezultatului, pe linii
      print*
      print*,'Matricea produs AxB este :'
      print*
      do i=1,m
       write(*,*) (prod(i,j),j=1,p)
      enddo
      end
```

6.3.4 Să se scrie un program FORTRAN care determină câte elemente sunt pozitive, negative, respectiv nule într-o matrice patrată $n \times n$.

Soluție propusă:

```
      program inventar
! --- determina cate nr. >0,<0,=0 exista intr-o matrice NxN reala
!
      implicit none
      real a(50,50)        ! dim maxima a matricii
      integer n            ! dimensiunea efectiva a matricii
      integer i,j,plus,minus,zero
      print*,'Dati dimensiunea efectiva a matricii A : N < 9'
      read*,n
! --- introducerea elem. matricii A
      print*
      print*,'       Dati elementele matricii A :'
      print*,'    ... primul indice        -> linia'
      print*,'    ... al doilea indice     -> coloana'
      print*
      do i=1,n
       do j=1,n
        write(*,'(1x,a2,i1,a1,i1,a2,$)') 'a(',i,',',j,')='
        read(*,*) a(i,j)
       enddo
      enddo
! --- initializare contori
      plus=0
      minus=0
      zero=0
! --- inventar dupa semnul elementelor
      do 4 i=1,n
       do 5 j=1,n
         if(a(i,j).gt.0.)then
           plus=plus+1
         elseif(a(i,j).lt.0.) then
           minus=minus+1
         else
           zero=zero+1
         endif
   5   continue
   4  continue
! --- afisarea rezultatului
      print*
      print*,' *** Rezultat inventar ***'
      print*
      print*,'Nr. pozitive : ',plus
      print*,'Nr. negative : ',minus
      print*,'Nr. nule : ',zero
      end
```

6.3.5 Să se scrie un program FORTRAN care determină câte elemente dintr-un şir de *n* numere naturale sunt divizibile cu 3.

Soluție propusă:

```
      program diviz3
! --- determina cate nr. divizibile cu 3 exista intr-un
!     sir de N nr. naturale
!
      implicit none
      integer a(50)          ! dim maxima a sirului
      integer n              ! dimensiunea efectiva a sirului
      integer i,k
11    print*,'dati nr. de elemente din sir : N < 50'
      read*,n
! --- verificare nr. elem. (utilizatorii pot sa nu fie atenti !!)
      if(n.lt.1.or.n.gt.50) goto 11
! --- introducerea elem. sirului
      print*
      print*,'Introduceti elem. sirului:'
      print*
      do i=1,n
        write(*,'(1x,a2,i2,a2,$)') 'a(',i,')='
        read(*,*) a(i)
      enddo
! --- initializare contor
      k=0
! --- determinarea nr. de elemente divizibile cu 3
      do i=1,n
       if(mod(a(i),3)==0) k=k+1
      enddo
! --- afisarea rezultatului
      write(*,'(//,a26,i2,a20)')'In sirul introdus exista ',k,
     +                          ' nr. DIVIZIBILE cu 3'
      end
```

6.3.6 Să se scrie un program FORTRAN care calculează media aritmetică a termenilor şirului de numere reale x_1, x_2, ..., x_n cuprinşi între *a* şi *b*, $a < b$ (reali).

Soluție propusă:

```
      program sirmedie
! --- calculeaza media elem. dintr-un sir, cuprinse intre A si B
!
      implicit none
      real x(50)            ! dim maxima a sirului
      integer n             ! dimensiunea efectiva a sirului
      integer i,k
      real a,b              ! intervalul [A,B]
      real suma,media
113   print*,'dati nr. de elemente din sir : N < 50'
      read*,n
! --- verificare nr. elem. (utilizatorii pot sa greseasca !!)
```

```
      if(n.lt.1.or.n.gt.50) goto 113
! --- introducerea elem. sirului
      print*
      print*,'Introduceti elem. sirului:'
      print*
      do i=1,n
        write(*,'(1x,a2,i2,a2,$)') 'x(',i,')='
        read(*,*) x(i)
      enddo
! --- introducerea capetelor intervalului de control [A,B]
888   print*
      print*,' Dati capetele intervalului [A,B]:'
      write(*,'(/,5x,a2,$)') 'A='
      read*,a
      write(*,'(/,5x,a2,$)') 'B='
      read*,b
! --- verificare A<B
      if(a.ge.b) goto 888
! --- initializare contor pt. elemente ; initializare suma
      k=0
      suma=0
! --- determinarea nr. de elemente care se gasesc in intervalul [A,B]
      do i=1,n
       if((x(i).ge.a).and.(x(i).le.b)) then
        k=k+1
        suma=suma+x(i)
       endif
      enddo
!
      media=suma/k
!
! --- afisarea rezultatului
      print*
      if(k.ne.0)then
        print*
        print*,'Media elem. din [A,B] este :',media
      else
        print*
        print*,'Nu exista nici un elem. al sirului in [A,B] !!'
        print*,'     deci media lor nu se poate calcula'
      endif
      end
```

6.3.7 Se introduce de la tastatură un şir de *n* numere întregi. Să se scrie un program FORTRAN care calculează produsul elementelor mai mici decât 100 şi suma celor mai mari decât 10.

Soluție propusă:

```
      program sirSP
! --- calculeaza suma, respectiv produsul elem. dintr-un sir
!     care satisfac anumite conditii
!
      implicit none
```

```
      integer a(30)        ! dim maxima a sirului
      integer n            ! dimensiunea efectiva a sirului
      integer i,ks,kp
      real suma,prod      ! produsul poate genera usor OVERFLOW in INTEGER
116   print*,'dati nr. de elemente din sir : N < 30'
      read*,n
! --- verificare nr. elem.
      if(n.lt.1.or.n.gt.30) goto 116
! --- introducerea elem. sirului
      print*
      print*,'Introduceti elem. sirului (nr. intregi !):'
      print*
      do i=1,n
        write(*,'(1x,a2,i2,a2,$)') 'a(',i,')='
        read(*,*) a(i)
      enddo
! --- initializare contori pt. elemente
      ks=0
      kp=0
! --- initializare suma si produs
      suma=0.
      prod=1.
! --- determinarea nr. de elemente pt. care calculez suma si prod
      do i=1,n
       if(a(i).gt.10) then
        ks=ks+1
        suma=suma+a(i)
       endif
       if(a(i).lt.100) then
        kp=kp+1
        prod=prod*a(i)
       endif
      enddo
! --- afisarea rezultatului
      print*
      if(ks.ne.0)then
        print*
        print*,'Suma elem. >10 este :',suma
      else
        print*
        print*,'Nu exista nici un elem. >10 in sir  !!'
        print*,'  deci suma lor nu se poate calcula'
      endif
!
      print*
      if(kp.ne.0)then
        print*
        print*,'Produsul elem. <100 este :',prod
      else
        print*
        print*,'Nu exista nici un elem. <100 in sir  !!'
        print*,' deci produsul lor nu se poate calcula'
      endif
      end
```

6.3.8 Să se scrie un program FORTRAN care generează transpusa unei matrici de dimensiuni *n*x*n*.

Soluție propusă:

```
      program transp
! --- transpusa unei matrici reale NxN
!
      implicit none
      real a(50,50),at(50,50)    ! dim maxima a matricilor
      integer n                  ! dim efectiva
      integer i,j
1     print*,'dati dimens. efectiva a matricii patrate A:  N < 50'
      read*,n
! --- verificare
      if(n.lt.1.or.n.gt.50) goto 1
! --- introducerea elem. matricii A, linie dupa linie
      print*
      print*,'     Dati elementele matricii A, pe linii :'
      print*,'...deci cate n valori despartite de spatiu sau virgula'
      print*,'                  apoi <Enter>'
      print*
      do i=1,n
        read(*,*) (a(i,j),j=1,n)
      enddo
! --- generarea matricii transpuse
      do 44 i=1,n
       do 55 j=1,n
        at(i,j)=a(j,i)
   55  continue
   44 continue
! --- afisarea rezultatului, pe linii
      print*
      print*,'Matricea transpusa este :'
      print*
      do i=1,n
       write(*,*) (at(i,j),j=1,n)
      enddo
      end
```

6.3.9 Să se scrie un program FORTRAN care determină elementul maxim dintr-un şir de *n* numere naturale.

Soluție propusă:

```
      program emaxim
! --- determina elem. MAXIM dintr-un sir de N nr. naturale
!
      implicit none
      integer a(50)         ! dim maxima a sirului
      integer n             ! dimensiunea efectiva a sirului
      integer i,emax
50    print*,'dati nr. de elemente din sir : N < 50'
```

```
      read*,n
! --- verificare
      if(n.lt.1.or.n.gt.50) goto 50
! --- introducerea elem. sirului
      print*
      print*,'Introduceti elem. sirului:'
      print*
      do i=1,n
        write(*,'(1x,a2,i2,a2,$)') 'a(',i,')='
        read(*,*) a(i)
      enddo
! --- initializare
      emax=a(1)
! --- determinarea elem. MAXIM
      do i=2,n
       if(emax.lt.a(i)) emax=a(i)
      enddo
! --- afisarea rezultatului
      print*
      print*,'Elementul maxim din sir este : ',emax
      print*
      end
```

6.3.10 Folosind problema precedentă, să se scrie un program FORTRAN care ordonează descrescător un şir de *n* numere naturale.

Soluție propusă:

```
      program ordsir
! --- ordoneaza descrescator un sir de N nr. naturale
!
      implicit none
      integer a(50)         ! dim maxima a sirului
      integer n             ! dimensiunea efectiva a sirului
      integer i,j,emax,idxmax
50    print*,'dati nr. de elemente din sir : N < 50'
      read*,n
! --- verificare
      if(n.lt.1.or.n.gt.50) goto 50
! --- introducerea elem. sirului
      print*
      print*,'Introduceti elem. sirului:'
      print*
      do i=1,n
        write(*,'(1x,a2,i2,a2,$)') 'a(',i,')='
        read(*,*) a(i)
      enddo
! --- ordonare
      do i=1,n-1
      emax=a(i)
      idxmax=i
      do j=i+1,n
       if(emax.lt.a(j)) then
        emax=a(j)
```

```
          idxmax=j
         endif
        enddo
        ! manevra SWAP
         a(idxmax)=a(i)
         a(i)=emax
        enddo
! --- afisarea rezultatului
      print*
      print*,'Sirul ordonat descrescator este : ',(a(i),i=1,n)
      print*
      end
```

6.3.11 Să se scrie un program FORTRAN care ordonează crescător un şir de *n* numere reale prin interschimbarea elementelor consecutive.

Soluţie propusă:

```
      program ordswap
! --- ordoneaza crescator un sir de N nr. reale prin interschimbare
!
      implicit none
      real a(30)         ! dim maxima a sirului
      integer n          ! dimensiunea efectiva a sirului
      integer i
      logical flag
      real manevr
30    print*,'dati nr. de elemente din sir : N < 31'
      read*,n
! --- verificare
      if(n.lt.1.or.n.gt.30) goto 30
! --- introducerea elem. sirului
      print*
      print*,'Introduceti elem. sirului:'
      print*
      do i=1,n
        write(*,'(1x,a2,i2,a2,$)') 'a(',i,')='
        read(*,*) a(i)
      enddo
! --- ordonare prin interschimbare
1000  flag=.true.     ! sirul este ordonat daca flag nu se mai modifica
      do i=1,n-1
       if(a(i).gt.a(i+1)) then
         flag=.false. ! adica sirul nu este inca ordonat
         ! urmeaza SWAP (interschimbare), cu variabila de manevra
         manevr=a(i+1)
         a(i+1)=a(i)
         a(i)=manevr
       endif
      enddo
      if(flag.eqv..false.) goto 1000
! --- afisarea rezultatului
      print*
      print*,'Sirul ordonat crescator este : ',(a(i),i=1,n)
```

```
      print*
      end
```

6.3.12 Să se scrie un program FORTRAN care ordonează elevii unei clase în ordinea descrescătoare a mediilor. Datele se vor introduce de la tastatură.

Soluție propusă:

```
      program clasa
! --- ordoneaza elevii unei clase in ordinea descrescatoare a mediilor
!
      implicit none
      character*30 nume(40) ! max 40 elevi, nume+pren. max 30 caractere
      real media(40)
      integer n             ! dimensiunea efectiva a clasei
      integer i,j,pozitia
      real medmax
      character*30 manevra
300   print*,'dati nr. de elevi din clasa : N < 41'
      read*,n
! --- verificare
      if(n.lt.1.or.n.gt.40) goto 300
! --- introducerea elem. sirului
      print*
      print*,'Introduceti elevii si mediile lor:'
      print*
      do i=1,n
        write(*,'(/,a7,i2,a3,$)') 'Elevul ',i,' : '
        read(*,'(a)') nume(i)  !format pentru lungime oarecare
         ! in executie, se vor introduce fara apostrof nume+prenume
        write(*,'(a10,a30,a3,$)') 'Media lui ',nume(i),' : '
        read*, media(i)
      enddo
! --- ordonare dupa medii
      do i=1,n-1
      medmax=media(i)
      pozitia=i
      do j=i+1,n
       if(medmax.lt.media(j)) then
        medmax=media(j)
        pozitia=j
       endif
      enddo
      ! manevra de interschimbare
       media(pozitia)=media(i)
       media(i)=medmax
      ! se deplaseaza si numele la noua pozitie
       manevra=nume(i)
       nume(i)=nume(pozitia)
       nume(pozitia)=manevra
      enddo
! --- afisarea rezultatului
      print '(//)'
      print*,'      Elevii ordonati in ordinea mediilor :'
```

```
      print*
      do i=1,n
       print '(1x,i2,a2,a30,a8,f5.2)',i,'. ',nume(i),' Media: ',media(i)
      enddo
      end
```

6.3.13 Să se scrie un program FORTRAN care determină cel mai mic element dintr-o matrice cu numere întregi, de dimensiuni $m \times n$.

Soluție propusă:

```
      program minmatr
! --- determina elem. MINIM dintr-o matrice MxN de nr. intregi
!
      implicit none
      integer a(40,30)    ! dim maxima a matricii
      integer m,n         ! dimensiunile efective ale matricii
      integer i,j,emin
1     print*,'dati dimensiunile efective ale matricii : M<40,N<30'
      read*,m,n
! --- verificare dimensiuni A
      if(m.lt.1.or.m.gt.40.or.n.lt.1.or.n.gt.30) goto 1
! --- introducerea elem. matricii A
      print*
      print*,'       Dati elementele matricii (nr. INTREGI):'
      print*,'    ... primul indice        -> linia'
      print*,'    ... al doilea indice     -> coloana'
      print*
      do i=1,m
       do j=1,n
        write(*,'(1x,a2,i1,a1,i1,a2,$)') 'a(',i,',',j,')='
        read(*,*) a(i,j)
       enddo
      enddo
! --- initializare
      emin=a(1,1)
! --- determinare MINIM
      do 4 i=1,m
       do 5 j=1,n
        if(a(i,j).lt.emin) emin=a(i,j)
   5   continue
   4  continue
! --- afisarea rezultatului
      print*
      print*,'Elementul MINIM al matricii este: ',emin
      end
```

6.3.14 Să se scrie un program FORTRAN care înlocuieşte elementul maxim de pe fiecare coloană a unei matrici *m*x*n* cu suma elementelor de pe coloana respectivă.

Soluție propusă:

```
      program sumcol
! --- se inlocuieste elem. MAXIM de pe fiecare coloana a unei
!     matrici reale MxN cu suma elem. de pe coloana respectiva
!
      implicit none
      real a(40,30)          ! dim maxima a matricii
      integer m,n            ! dim efectiva
      integer i,j,idx
      real emax,suma
1     print*,'dati dimensiunile efective ale matricii : M<40,N<30'
      read*,m,n
! --- verificare dimensiuni A
      if(m.lt.1.or.m.gt.40.or.n.lt.1.or.n.gt.30) goto 1
! --- introducerea elem. matricii A, linie dupa linie
      print*
      print*,'     Dati elementele matricii A, pe linii :'
      print*,'...deci cate n valori despartite de spatiu sau virgula'
      print*,'                  apoi <Enter>'
      print*
      do i=1,m
        read(*,*) (a(i,j),j=1,n)
      enddo
! --- prelucrare
      do 44 j=1,n         ! este convenabila parcurgerea pe coloane
       emax=a(1,j)
       idx=1
       suma=a(1,j)
       do 55 i=2,m
        suma=suma+a(i,j)
        if(a(i,j).gt.emax) then
         emax=a(i,j)
         idx=i
        endif
   55  continue
       a(idx,j)=suma      ! dar pentru mai multe maxime egale ?
                          ! studiati acest caz particular si
                          ! modificati programul
   44 continue
! --- afisarea rezultatului, pe linii
      print '(//)'
      print*,'Matricea modificata este :'
      print*
      do i=1,m
       write(*,*) (a(i,j),j=1,n)
      enddo
      end
```

6.4 EXERCIȚII CU SUBPROGRAME

6.4.1 Să se scrie un program FORTRAN care calculează suma 1!+2!+3!+...+10!. Se va utiliza un subprogram de tip FUNCTION pentru calculul lui *n*!.

Soluție propusă:

```
      program sumfact
! --- calculul sumei factorialelor primelor 10 nr. naturale
!     utilizand FUNCTION
!
      implicit none
      integer n
      parameter (n=10) !in F90, s-ar fi scris INTEGER,PARAMETER::n=10
      integer fact,s,i
      s=0
      do 55 i=1,n
       s=s+fact(i)
55    continue
      write(*,*) 'suma factorialelor primelor ',n,
     &           ' nr. naturale este : ',s
      end

      integer function fact(n)
! mai corect ar fi REAL function, caci exista pericol de Overflow
      integer n,j,i
      j=1
      do i=1,n
       j=j*i
      enddo
      fact=j
      return
      end
```

6.4.2 Să se scrie un program FORTRAN care calculează perimetrul şi aria unui patrat de latura *L*, în 3 variante: folosind funcție definită aritmetic, funcție definită ca modul (FUNCTION), respectiv subprogram (SUBROUTINE).

Soluție propusă (utilizând funcție definită aritmetic):

```
      program patrat1
! --- calculul perimetrului si ariei unui patrat de latura L
!     utilizand o functie definita aritmetic
!
      implicit none
      real aria,perim,latura
      real a,p,l
! --- aici se definesc aritmetic 2 functii cu parametrul formal LATURA
      perim(latura)=4*latura
      aria(latura)=latura**2
!
      print*, 'Dati latura patratului :'
```

```
      read*,l
! --- la apel, se inlocuieste parametrul formal cu cel efectiv L
!     rezultatul este returnat prin intermediul numelui functiei
      p=perim(l)
      a=aria(l)
! --- afisare rezultate
      print*
      print*
      write(*,*) 'Perimetrul patratului de latura ',l,' este : ',p
      write(*,*) 'Aria patratului de latura ',l,' este : ',a
      end
```

Soluție propusă (utilizând FUNCTION):

```
      program patrat2
! --- calculul perimetrului si ariei unui patrat de latura L
!     utilizand FUNCTION
!
      implicit none
      real aria,perim
      real p,a,l
      print*, 'Dati latura patratului :'
      read*,l
! --- la apel, se inlocuieste parametrul formal cu cel efectiv L
!     rezultatul este returnat prin intermediul numelui functiei
      p=perim(l)
      a=aria(l)
! --- afisare rezultate
      print*
      print*
      write(*,*) 'Perimetrul patratului de latura ',l,' este : ',p
      write(*,*) 'Aria patratului de latura ',l,' este : ',a
      end

      real function perim(latura)
      real latura
      perim=4*latura
      end

      real function aria(latura)
      real latura
      aria=latura**2
      end
```

Soluție propusă (utilizând SUBROUTINE):

```
      program patrat3
! --- calculul perimetrului si ariei unui patrat de latura L
!     utilizand SUBROUTINE
!
      implicit none
      real l,p,a
      print*, 'Dati latura patratului :'
      read*,l
! --- la apel, se inlocuiesc parametrii formali cu cei efectivi
```

```
!     parametrul de intrare este l
!     rezultatul este returnat prin parametrii p,a (de iesire)
      call calcul(l,p,a)
! --- afisare rezultate
      print*
      print*
      write(*,*) 'Perimetrul patratului de latura ',l,' este : ',p
      write(*,*) 'Aria patratului de latura ',l,' este : ',a
      end

      subroutine calcul(latura,perim,aria)
      real latura,perim,aria
      perim=4*latura
      aria=latura**2
      end
```

6.4.3 Să se scrie un program FORTRAN care transformă un număr din baza 2 în baza 10, folosind un subprogram de tip SUBROUTINE.

Soluție propusă:

```
      program trans2_10
! --- converteste un nr. din baza 2 in baza 10
!     utilizand SUBROUTINE
!
      implicit none
      character*20 nrbin
      integer nrzece
      write(*,*)'Transformarea unui nr. din baza 2 in baza 10'
666   write(*,*)
      write(*,*) 'Dati nr. binar (max 20 cifre, doar 0 si 1): '
      read(*,'(a)')nrbin
      call tr210(nrbin,nrzece)
      if(nrzece.eq.-1) goto 666     ! input gresit
      write(*,*) 'Nr. in baza 10 este : ',nrzece
      end

      subroutine tr210(doi,zece)
      character*(*) doi
      integer zece
      integer cifre,i,aport
! --- initializare
      cifre=0
      zece=0
! --- determinare nr. de cifre
      do i=len(doi),1,-1  ! LEN determina lungimea var. tip caracter
       if(doi(i:i).ne.' ') goto 888      ! test subsir de 1 caracter
      enddo
888   cifre=i
! --- validare cifre (in nr. binar sa fie numai 0 sau 1 !!)
      if(cifre.eq.0.or.doi(1:1).eq.'0') then
        print*, ' Numar gresit ! '
        zece=-1
        goto 77
```

```
      endif
! --- transformare
      do i=1,cifre
         if(doi(i:i).eq.'0') then
          aport=0
         elseif(doi(i:i).eq.'1') then
          aport=2**(cifre-i)
          zece=zece+aport
         else
          print*, ' Numar gresit ! '
          zece=-1
          exit    ! iesire fortata din DO
         endif
       enddo
77     return
       end
```

6.4.4 Să se scrie un program FORTRAN care calculează cel mai mare divizor comun a două numere naturale, pe baza algoritmului lui Euclid. Se va utiliza un subprogram de tip FUNCTION.

Soluție propusă:

```
      program divcom
! --- determina CMMDC a 2 nr. naturale nenule
!     utilizand subprogram FUNCTION
!
      implicit none
      integer a,b
      integer cmmdc
44    print*,'Dati primul nr. natural : '
      read*,a
      if(a.le.0) then
       print*, 'NUMAR GRESIT !'
       goto 44
      endif
55    write(*,*) 'Dati al doilea nr. natural : '
      read(*,*)b
      if(b.le.0) then
       print*, 'NUMAR GRESIT !'
       goto 55
      endif
      print '(//)'
      write(*,*) 'Cel mai mare divizor comun este : ',cmmdc(a,b)
      end

      integer function cmmdc(nr1,nr2)
      integer nr1,nr2,x,y,rest
      x=nr1
      y=nr2
      rest=mod(x,y)
1     if(rest.ne.0) then
       x=y
       y=rest
```

```
        rest=mod(x,y)
        goto 1
       else
        cmmdc=y
       endif
       end
```

6.4.5 Să se scrie un program FORTRAN care afişează toate numerele prime mai mici decât 1000. Se va utiliza un subprogram de tip SUBROUTINE.

Soluție propusă:

```
      program nrprime
! --- determina toate nr. prime mai mici decat 1000
!     folosind subprogram SUBROUTINE
!
      implicit none
      integer limita
      parameter (limita=1000)
      print*, ' Nr. prime de la 1 la ',limita,' sunt : '
      print*
      print '(1x,i3,1x,i3,1x,$)',1,2       ! primele 2 nr. prime
      call prim(limita)
      end

      subroutine prim(limsup)
      integer limsup,i,j,k
      logical iprim
      do i=3,limsup,2  ! nr. prime sunt o submultime a celor impare
       iprim=.true.
       j=int(sqrt(i*1.))      !  SQRT cere argument real
                       ! j este limita pana la care caut divizori
       do k=2,j
        if(mod(i,k)==0) then
         iprim=.false.
         exit          ! iesire fortata din DO
        endif
       enddo
       if(iprim.eqv..true.) then
        write(*,'(1x,i3,$)')i
       endif
      enddo
      end
```

6.4.6 Să se scrie un program FORTRAN care determină câtul şi restul împărțirii unui polinom de grad *n* la binomul (*X*-*a*), folosind schema lui Horner. Se va utiliza un subprogram de tip SUBROUTINE.

Soluție propusă:

```
      program polinom
! --- determina catul si restul impartirii unui polinom P(x) la (X-a)
!     folosind subprogram SUBROUTINE pt. schema lui Horner
!
      implicit none
      real a(0:30)   ! sirul coeficientilor polinomului P(X)
      real b(0:29)   ! sirul coeficientilor catului Q(X)
      integer n      ! gradul polinomului P(X), n<30
      real rest,alfa
      integer i
90    print '(a35,$)', ' Dati gradul polinomului P (n<30): '
      read*,n
      if(n.lt.1.or.n.gt.30) goto 90
      print*
      print*,'Dati coef. lui P in ordine descr. a puterilor lui X :'
      print*
      do i=n,0,-1
       print '(a15,i2,a8,$)','- coef. lui X**',i,' este : '
       read*,a(i)
      enddo
      print*
      print '(a44,$)', ' Dati coeficientul "a" al binomului (X-a) : '
      read*,alfa
! --- apel
      call horner(a,b,n,alfa,rest)
! --- afisare rezultat
      print '(//)'
      print*,'   ************  REZULTAT  **************'
      print*
      print*,'Coef. catului, in ordine descr. a puterilor lui X :'
      print*
      do i=n-1,0,-1
       print '(a15,i2,a8,f7.3)','- coef. lui X**',i,' este : ',b(i)
      enddo
      print*
      print*, 'Restul impartirii lui P la (X-',alfa,') este :',rest
      end

      subroutine horner(p,q,gradp,a,r)
      integer gradp,i
      real p(0:gradp),q(0:(gradp-1))
      real a,r
      q(gradp-1)=p(gradp)
      do i=gradp-2,0,-1
        q(i)=q(i+1)*a+p(i+1)
      enddo
      r=q(0)*a+p(0)
      end
```

6.5 EXERCIȚII CU INTRĂRI/IEŞIRI FOLOSIND FIŞIERE

6.5.1 Să se modifice soluția propusă pentru problema **6.2.1**, folosind un fişier de ieşire pentru rezultate, în locul afişării pe ecran. Fişierul creat se va numi “L51.REZ”.

Soluție propusă:

```
      program expresie
! --- calculul valorii unei expresii in functie
!     de 4 variabile care parcurg un interval de nr. intregi
!     foloseste FISIER de iesire pentru rezultate
!
      implicit none
      integer a,b,c,d
      integer i   !contor al nr. de operatii/linii
      real e
!     initializare contor
      i=0
!     deschidere fisier pentru rezultate
      open(1,file='l51.rez',status='unknown')
!     4 cicluri DO
      do 1 a=0,2
       do 2 b=0,2
        do 3 c=0,2
         do 4 d=0,2
          i=i+1
! --- urmeaza acum un test de numitor nul
          if(c.eq.d) then
           write(1,456) 'a=',a,' b=',b,' c=',c,' d=',d,
     &                  '    e nu se poate calcula'
          else
           e=(a+2.0*b)/(c-d)
           write(1,567) 'a=',a,' b=',b,' c=',c,' d=',d,'    -> e=',e
          endif
4        continue
3       continue
2      continue
1     continue
!     inchidere fisier
      close(1)
! --- se afiseaza (pe ecran) cite operatii s-au efectuat
      write(*,'(//,1x,a14,i2,a9)') 's-au efectuat ',i,' operatii'
456   format(a2,i1,a3,i1,a3,i1,a3,i1,a25)
567   format(a2,i1,a3,i1,a3,i1,a3,i1,a9,f7.3)
      end
```

Notă: Conținutul fişierului de rezultate “L51.REZ” se găseşte în anexa **A-1**.

6.5.2 Să se modifice soluția propusă pentru problema **6.2.5**, folosind un fişier de ieşire pentru rezultate, în locul afişării pe ecran. Fişierul rezultat se va numi “L52.REZ”.

Soluție propusă:

```
      program functie
! --- calculeaza valorile functiei x**2+sin(x) in [0,2]
!     cu pasul 0.1
!     foloseste FISIER de iesire pentru rezultate
!
      implicit none
      real x,y
      integer i
!     deschidere fisier de iesire
      open(2,file='l52.rez',status='unknown')
!     ciclul DO
      do i=0,20,1
! este recomandabil a utiliza contor de tip intreg in ciclul DO
! chiar daca F90 accepta si contor nr. real, rezultatele pot fi
! eronate
        x=i/10.
        y=x*x+sin(x)
        write(2,'(a9,f3.1,a7,f8.6)')'pentru x=',x,'  f(x)=',y
! mai simplu, se putea scrie neformatat           PRINT*,x,y
      enddo
!     inchidere fisier
      close(2)
      end
```

Notă: Conținutul fişierului “L52.REZ” cu rezultate se găseşte în anexa **A-2**.

6.5.3 Să se modifice soluția propusă pentru problema **6.3.3**, folosind un fişier de intrare pentru cele două matrici şi un fisier de ieşire pentru rezultat. Fişierul cu datele de intrare se va numi “L53.DAT” iar cel cu rezultate “L53.REZ”.

Soluție propusă:

```
      program produs
! --- produsul a 2 matrici reale AxB
!     A este de tip MxN
!     B este de tip NxP
!     foloseste un FISIER pentru input (cele 2 matrici)
!               un FISIER pentru output (matricea produs)
!
      implicit none
      real a(50,40),b(40,60),prod(50,60)    ! dim maxima a matricilor
      integer m,n,n2,p                      ! dimensiunile efective
      integer i,j,k
      character*26 separa
! --- deschidere fisier de date
      open(1,file='l53.dat',status='old')
! --- citire dimens. efective ale matricii A
```

```
      read(1,*) m,n
! --- verificare dimensiuni A
      if(m.lt.1.or.m.gt.50.or.n.lt.1.or.n.gt.40) goto 1
! --- citire dimens. efective ale matricii B
      read(1,*) n2,p
! --- verificare compatibilitate
      if(n.ne.n2) then
       print*,'Inmultirea nu poate fi facuta ! Date ERONATE'
       stop
      endif
! --- verificare dimensiuni B
      if(p.lt.1.or.p.gt.60) goto 2
! --- citire separator
      read(1,*) separa
! --- citire elem. matricii A, linie cu linie
      do i=1,m
        read(1,*) (a(i,j),j=1,n)
      enddo
! --- citire separator
      read(1,*) separa
! --- citire elem. matricii B, linie cu linie
      do i=1,n
        read(1,*) (b(i,j),j=1,p)
      enddo
! --- inchidere fisier de intrare date
      close(1)
! --- inmultirea propriu-zisa
      do 44 i=1,m
       do 55 j=1,p
        prod(i,j)=0.
        do 66 k=1,n
         prod(i,j)=prod(i,j)+ a(i,k)*b(k,j)
   66   continue
   55  continue
   44 continue
! --- deschidere fisier de iesire (rezultat=matricea produs)
      open(2,file='l53.rez',status='unknown')
! --- afisarea rezultatului, pe linii
      write(2,*)
      write(2,*)'Matricea produs AxB este :'
      write(2,*)separa
      write(2,*)
      do i=1,m
       write(2,*) (prod(i,j),j=1,p)
      enddo
! --- inchidere fisier de rezultate
      close(2)
      stop
! --- tratarea erorilor din fisierul de intrare
1     print*,'EROARE DATE : dim. efective pt. matr. A vor fi M<50,N<40'
      stop
2     print*,'EROARE DATE : dim. efective pt. matr. B vor fi N<40,P<60'
      stop
      end
```

Notă: În anexa **A-3** este prezentat un exemplu pentru conținutul fişierului de date “L53.DAT” urmat de conținutul fişierului rezultat “L53.REZ” corespunzător.

6.5.4 Să se modifice soluția propusă pentru problema **6.4.5**, folosind un fişier de ieşire numit "T54.REZ" pentru rezultate, în condițiile în care trebuie determinate toate numerele prime până la 10000.

Soluție propusă:

```
      program prim_fisier
! --- determina toate nr. prime mai mici decat 10000
!     si scrie rezultatul intr-un fisier
!
      implicit none
      integer limita,fis
      parameter (limita=10000)
      parameter (fis=1)
      open(fis,file='t54.rez',status='unknown')
      write(fis,*) ' Nr. prime de la 1 la ',limita,' sunt : '
      write(fis,*)
      write(fis,'(1x,i5,1x,i5,1x,$)') 1,2    ! primele 2 nr. prime
      call prim(limita,fis)
      close(fis)
      end

      subroutine prim(limsup,fisier)
      integer limsup,i,j,k,fisier
      logical iprim
      do i=3,limsup,2    ! nr. prime sunt o submultime a celor impare
       iprim=.true.
       j=int(sqrt(i*1.))  ! SQRT cere argument real
       ! j este limita pana la care caut divizori
       do k=2,j
        if(mod(i,k)==0) then
         iprim=.false.
         exit   ! iesire fortata din DO
        endif
       enddo
       if(iprim.eqv..true.) then
        write(fisier,'(1x,i5,$)')i
       endif
      enddo
      end
```

Notă: Conținutul fişierului de rezultate "T54.REZ" este prezentat în anexa **A-4**.

6.5.5 Să se modifice soluția propusă pentru problema **6.3.12**, folosind un fişier de intrare numit "T55.DAT" pentru date şi un fisier de ieşire numit "T55.REZ" pentru rezultate.

Soluție propusă:

```
      program clasa_fisier
! --- ordoneaza elevii unei clase in ordinea descrescatoare a mediilor
!     utilizand fisiere pt. input, respectiv output
```

```
      implicit none
      integer nmax
      parameter (nmax=40)
      character*30 nume(nmax) !max NMAX elevi, nume+pren. max 30 caract.
      real media(nmax)
      integer n    ! dimensiunea efectiva a clasei
      integer i,j,pozitia
      real medmax
      character*30 manevra
      character*1 separator
! --- initializare nr. inregistrari in fisier
      n=0
! --- citirea datelor din fisier
      open(1,file='t55.dat',status='old')
788    read(1,'(a30,a1,f5.2)',END=500)nume(n+1),separator,media(n+1)
       ! daca se atinge END OF FILE se continua cu CLOSE(1)
       n=n+1
       ! verificare nr. elevi
       if(n.gt.nmax) then
        print*,'EROARE : fisierul contine mai mult de 40 nume elevi !'
        stop !gata
       else
        goto 788  !se continua cu un nou elev
       endif
500   close(1)
! --- ordonare dupa medii
      do i=1,n-1
      medmax=media(i)
      pozitia=i
      do j=i+1,n
       if(medmax.lt.media(j)) then
        medmax=media(j)
        pozitia=j
       endif
      enddo
       ! manevra de interschimbare
       media(pozitia)=media(i)
       media(i)=medmax
       ! se deplaseaza si numele la noua pozitie
       manevra=nume(i)
       nume(i)=nume(pozitia)
       nume(pozitia)=manevra
      enddo
! --- fisierul de rezultate
      open(2,file='t55.rez',status='unknown')
       write(2,'(//)')
       write(2,*)'      Elevii ordonati in ordinea mediilor :'
       write(2,'(//)')
       do i=1,n
        write(2,44)i,'. ',nume(i),' Media: ',media(i)
44      format(1x,i2,a2,a30,a8,f5.2)
       enddo
      close(2)
      end
```

Notă: În anexa **A-5** se prezintă conţinutul fişierelor “T55.DAT” şi “T55.REZ”.

6.6 EXERCIȚII DIVERSE

6.6.1 Să se scrie un program FORTRAN care simulează aruncarea simultană a 2 zaruri.

Soluție propusă:

```
      program zaruri
! --- simularea aruncarii simultane a 2 zaruri
!
      implicit none
      real r(2)
      integer zar(2),i
!initializare generator de numere pseudoaleatoare
      r(1) = RAND(TIME())
!
      do i=1,2
        r(i) = RAND(0)
        zar(i)=int(6*r(i))+1
        print*,zar(i)
      enddo
      end

! Varianta scrisa in Fortran 90 standard
!
!     program zaruri
!     implicit none
!     real,dimension(2)::r
!     integer,dimension(2)::zar
!     integer::i
!     call random_seed()
!     call random_number(r)
!     do i=1,2
!       zar(i)=int(6*r(i))+1
!       print*,zar(i)
!     enddo
!     end program zaruri
```

6.6.2 Să se scrie un program FORTRAN cu ajutorul căruia să se simuleze o extragere LOTO 6/49.

Soluție propusă:

```
      program loto
! --- simuleaza o extragere LOTO 6/49
!
      implicit none
      integer nmax,cite,idx,i,manevra
      parameter(nmax=49)
      parameter(cite=6)
      integer sir(nmax)
      real r(nmax)
! initializare generator de numere pseudoaleatoare
```

```
      r(1)=rand(time())
! initializare sir de numere intregi de la 1 la 49
      do i=1,nmax
        sir(i) = i
      enddo
! extragere (fara repetarea vreunui nr. !)
      do i=1,cite
        r(i)=rand(0)
        idx = INT(r(i) * (nmax - i + 1)) + i
         manevra=sir(i)
         sir(i)=sir(idx)
         sir(idx)=manevra
        print*, sir(i)
      enddo
      end
```

6.7 ANEXE

A-1 Conținutul fișierului de rezultate "L51.REZ" de la exercițiul 6.5.1:

```
a=0 b=0 c=0 d=0    e nu se poate calcula
a=0 b=0 c=0 d=1    -> e=  0.000
a=0 b=0 c=0 d=2    -> e=  0.000
a=0 b=0 c=1 d=0    -> e=  0.000
a=0 b=0 c=1 d=1    e nu se poate calcula
a=0 b=0 c=1 d=2    -> e=  0.000
a=0 b=0 c=2 d=0    -> e=  0.000
a=0 b=0 c=2 d=1    -> e=  0.000
a=0 b=0 c=2 d=2    e nu se poate calcula
a=0 b=1 c=0 d=0    e nu se poate calcula
a=0 b=1 c=0 d=1    -> e= -2.000
a=0 b=1 c=0 d=2    -> e= -1.000
a=0 b=1 c=1 d=0    -> e=  2.000
a=0 b=1 c=1 d=1    e nu se poate calcula
a=0 b=1 c=1 d=2    -> e= -2.000
a=0 b=1 c=2 d=0    -> e=  1.000
a=0 b=1 c=2 d=1    -> e=  2.000
a=0 b=1 c=2 d=2    e nu se poate calcula
a=0 b=2 c=0 d=0    e nu se poate calcula
a=0 b=2 c=0 d=1    -> e= -4.000
a=0 b=2 c=0 d=2    -> e= -2.000
a=0 b=2 c=1 d=0    -> e=  4.000
a=0 b=2 c=1 d=1    e nu se poate calcula
a=0 b=2 c=1 d=2    -> e= -4.000
a=0 b=2 c=2 d=0    -> e=  2.000
a=0 b=2 c=2 d=1    -> e=  4.000
a=0 b=2 c=2 d=2    e nu se poate calcula
a=1 b=0 c=0 d=0    e nu se poate calcula
a=1 b=0 c=0 d=1    -> e= -1.000
a=1 b=0 c=0 d=2    -> e= -0.500
a=1 b=0 c=1 d=0    -> e=  1.000
a=1 b=0 c=1 d=1    e nu se poate calcula
a=1 b=0 c=1 d=2    -> e= -1.000
a=1 b=0 c=2 d=0    -> e=  0.500
a=1 b=0 c=2 d=1    -> e=  1.000
a=1 b=0 c=2 d=2    e nu se poate calcula
a=1 b=1 c=0 d=0    e nu se poate calcula
a=1 b=1 c=0 d=1    -> e= -3.000
a=1 b=1 c=0 d=2    -> e= -1.500
a=1 b=1 c=1 d=0    -> e=  3.000
a=1 b=1 c=1 d=1    e nu se poate calcula
a=1 b=1 c=1 d=2    -> e= -3.000
a=1 b=1 c=2 d=0    -> e=  1.500
a=1 b=1 c=2 d=1    -> e=  3.000
a=1 b=1 c=2 d=2    e nu se poate calcula
a=1 b=2 c=0 d=0    e nu se poate calcula
a=1 b=2 c=0 d=1    -> e= -5.000
a=1 b=2 c=0 d=2    -> e= -2.500
a=1 b=2 c=1 d=0    -> e=  5.000
a=1 b=2 c=1 d=1    e nu se poate calcula
a=1 b=2 c=1 d=2    -> e= -5.000
a=1 b=2 c=2 d=0    -> e=  2.500
```

```
a=1 b=2 c=2 d=1    -> e=  5.000
a=1 b=2 c=2 d=2    e nu se poate calcula
a=2 b=0 c=0 d=0    e nu se poate calcula
a=2 b=0 c=0 d=1    -> e= -2.000
a=2 b=0 c=0 d=2    -> e= -1.000
a=2 b=0 c=1 d=0    -> e=  2.000
a=2 b=0 c=1 d=1    e nu se poate calcula
a=2 b=0 c=1 d=2    -> e= -2.000
a=2 b=0 c=2 d=0    -> e=  1.000
a=2 b=0 c=2 d=1    -> e=  2.000
a=2 b=0 c=2 d=2    e nu se poate calcula
a=2 b=1 c=0 d=0    e nu se poate calcula
a=2 b=1 c=0 d=1    -> e= -4.000
a=2 b=1 c=0 d=2    -> e= -2.000
a=2 b=1 c=1 d=0    -> e=  4.000
a=2 b=1 c=1 d=1    e nu se poate calcula
a=2 b=1 c=1 d=2    -> e= -4.000
a=2 b=1 c=2 d=0    -> e=  2.000
a=2 b=1 c=2 d=1    -> e=  4.000
a=2 b=1 c=2 d=2    e nu se poate calcula
a=2 b=2 c=0 d=0    e nu se poate calcula
a=2 b=2 c=0 d=1    -> e= -6.000
a=2 b=2 c=0 d=2    -> e= -3.000
a=2 b=2 c=1 d=0    -> e=  6.000
a=2 b=2 c=1 d=1    e nu se poate calcula
a=2 b=2 c=1 d=2    -> e= -6.000
a=2 b=2 c=2 d=0    -> e=  3.000
a=2 b=2 c=2 d=1    -> e=  6.000
a=2 b=2 c=2 d=2    e nu se poate calcula
```

A-2 Conținutul fişierului de rezultate “L52.REZ” de la exercițiul 6.5.2:

```
pentru x=0.0  f(x)=0.000000
pentru x=0.1  f(x)=0.109833
pentru x=0.2  f(x)=0.238669
pentru x=0.3  f(x)=0.385520
pentru x=0.4  f(x)=0.549418
pentru x=0.5  f(x)=0.729426
pentru x=0.6  f(x)=0.924643
pentru x=0.7  f(x)=1.134218
pentru x=0.8  f(x)=1.357356
pentru x=0.9  f(x)=1.593327
pentru x=1.0  f(x)=1.841471
pentru x=1.1  f(x)=2.101207
pentru x=1.2  f(x)=2.372039
pentru x=1.3  f(x)=2.653558
pentru x=1.4  f(x)=2.945450
pentru x=1.5  f(x)=3.247495
pentru x=1.6  f(x)=3.559574
pentru x=1.7  f(x)=3.881665
pentru x=1.8  f(x)=4.213847
pentru x=1.9  f(x)=4.556300
pentru x=2.0  f(x)=4.909297
```

A-3 Pentru exercițiul 6.5.3:

Exemplu de fişier cu date (L53.DAT) pentru exercițiul 6.5.3:

```
2,2
2,3
===================
1 2
3 4
===================
5 6 7
8 9 5
```

Conținutul fişierului cu rezultate (L53.REZ) de la exercițiul 6.5.3:

```
 Matricea produs AxB este :
 ===================

  21.  24.  17.
  47.  54.  41.
```

A-4 Conținutul fişierului de rezultate "T54.REZ" de la exercițiul 6.5.4:

```
 Nr. prime de la 1 la  10000 sunt :

 1     2     3     5     7     11    13    17    19    23    29    31    37
41    43    47    53    59    61    67    71    73    79    83    89    97
101   103   107   109   113   127   131   137   139   149   151   157   163
167   173   179   181   191   193   197   199   211   223   227   229   233
239   241   251   257   263   269   271   277   281   283   293   307   311
313   317   331   337   347   349   353   359   367   373   379   383   389
397   401   409   419   421   431   433   439   443   449   457   461   463
467   479   487   491   499   503   509   521   523   541   547   557   563
569   571   577   587   593   599   601   607   613   617   619   631   641
643   647   653   659   661   673   677   683   691   701   709   719   727
733   739   743   751   757   761   769   773   787   797   809   811   821
823   827   829   839   853   857   859   863   877   881   883   887   907
911   919   929   937   941   947   953   967   971   977   983   991   997
1009  1013  1019  1021  1031  1033  1039  1049  1051  1061  1063  1069  1087
1091  1093  1097  1103  1109  1117  1123  1129  1151  1153  1163  1171  1181
1187  1193  1201  1213  1217  1223  1229  1231  1237  1249  1259  1277  1279
1283  1289  1291  1297  1301  1303  1307  1319  1321  1327  1361  1367  1373
1381  1399  1409  1423  1427  1429  1433  1439  1447  1451  1453  1459  1471
1481  1483  1487  1489  1493  1499  1511  1523  1531  1543  1549  1553  1559
1567  1571  1579  1583  1597  1601  1607  1609  1613  1619  1621  1627  1637
1657  1663  1667  1669  1693  1697  1699  1709  1721  1723  1733  1741  1747
1753  1759  1777  1783  1787  1789  1801  1811  1823  1831  1847  1861  1867
1871  1873  1877  1879  1889  1901  1907  1913  1931  1933  1949  1951  1973
1979  1987  1993  1997  1999  2003  2011  2017  2027  2029  2039  2053  2063
2069  2081  2083  2087  2089  2099  2111  2113  2129  2131  2137  2141  2143
2153  2161  2179  2203  2207  2213  2221  2237  2239  2243  2251  2267  2269
2273  2281  2287  2293  2297  2309  2311  2333  2339  2341  2347  2351  2357
2371  2377  2381  2383  2389  2393  2399  2411  2417  2423  2437  2441  2447
```

2459	2467	2473	2477	2503	2521	2531	2539	2543	2549	2551	2557	2579
2591	2593	2609	2617	2621	2633	2647	2657	2659	2663	2671	2677	2683
2687	2689	2693	2699	2707	2711	2713	2719	2729	2731	2741	2749	2753
2767	2777	2789	2791	2797	2801	2803	2819	2833	2837	2843	2851	2857
2861	2879	2887	2897	2903	2909	2917	2927	2939	2953	2957	2963	2969
2971	2999	3001	3011	3019	3023	3037	3041	3049	3061	3067	3079	3083
3089	3109	3119	3121	3137	3163	3167	3169	3181	3187	3191	3203	3209
3217	3221	3229	3251	3253	3257	3259	3271	3299	3301	3307	3313	3319
3323	3329	3331	3343	3347	3359	3361	3371	3373	3389	3391	3407	3413
3433	3449	3457	3461	3463	3467	3469	3491	3499	3511	3517	3527	3529
3533	3539	3541	3547	3557	3559	3571	3581	3583	3593	3607	3613	3617
3623	3631	3637	3643	3659	3671	3673	3677	3691	3697	3701	3709	3719
3727	3733	3739	3761	3767	3769	3779	3793	3797	3803	3821	3823	3833
3847	3851	3853	3863	3877	3881	3889	3907	3911	3917	3919	3923	3929
3931	3943	3947	3967	3989	4001	4003	4007	4013	4019	4021	4027	4049
4051	4057	4073	4079	4091	4093	4099	4111	4127	4129	4133	4139	4153
4157	4159	4177	4201	4211	4217	4219	4229	4231	4241	4243	4253	4259
4261	4271	4273	4283	4289	4297	4327	4337	4339	4349	4357	4363	4373
4391	4397	4409	4421	4423	4441	4447	4451	4457	4463	4481	4483	4493
4507	4513	4517	4519	4523	4547	4549	4561	4567	4583	4591	4597	4603
4621	4637	4639	4643	4649	4651	4657	4663	4673	4679	4691	4703	4721
4723	4729	4733	4751	4759	4783	4787	4789	4793	4799	4801	4813	4817
4831	4861	4871	4877	4889	4903	4909	4919	4931	4933	4937	4943	4951
4957	4967	4969	4973	4987	4993	4999	5003	5009	5011	5021	5023	5039
5051	5059	5077	5081	5087	5099	5101	5107	5113	5119	5147	5153	5167
5171	5179	5189	5197	5209	5227	5231	5233	5237	5261	5273	5279	5281
5297	5303	5309	5323	5333	5347	5351	5381	5387	5393	5399	5407	5413
5417	5419	5431	5437	5441	5443	5449	5471	5477	5479	5483	5501	5503
5507	5519	5521	5527	5531	5557	5563	5569	5573	5581	5591	5623	5639
5641	5647	5651	5653	5657	5659	5669	5683	5689	5693	5701	5711	5717
5737	5741	5743	5749	5779	5783	5791	5801	5807	5813	5821	5827	5839
5843	5849	5851	5857	5861	5867	5869	5879	5881	5897	5903	5923	5927
5939	5953	5981	5987	6007	6011	6029	6037	6043	6047	6053	6067	6073
6079	6089	6091	6101	6113	6121	6131	6133	6143	6151	6163	6173	6197
6199	6203	6211	6217	6221	6229	6247	6257	6263	6269	6271	6277	6287
6299	6301	6311	6317	6323	6329	6337	6343	6353	6359	6361	6367	6373
6379	6389	6397	6421	6427	6449	6451	6469	6473	6481	6491	6521	6529
6547	6551	6553	6563	6569	6571	6577	6581	6599	6607	6619	6637	6653
6659	6661	6673	6679	6689	6691	6701	6703	6709	6719	6733	6737	6761
6763	6779	6781	6791	6793	6803	6823	6827	6829	6833	6841	6857	6863
6869	6871	6883	6899	6907	6911	6917	6947	6949	6959	6961	6967	6971
6977	6983	6991	6997	7001	7013	7019	7027	7039	7043	7057	7069	7079
7103	7109	7121	7127	7129	7151	7159	7177	7187	7193	7207	7211	7213
7219	7229	7237	7243	7247	7253	7283	7297	7307	7309	7321	7331	7333
7349	7351	7369	7393	7411	7417	7433	7451	7457	7459	7477	7481	7487
7489	7499	7507	7517	7523	7529	7537	7541	7547	7549	7559	7561	7573
7577	7583	7589	7591	7603	7607	7621	7639	7643	7649	7669	7673	7681
7687	7691	7699	7703	7717	7723	7727	7741	7753	7757	7759	7789	7793
7817	7823	7829	7841	7853	7867	7873	7877	7879	7883	7901	7907	7919
7927	7933	7937	7949	7951	7963	7993	8009	8011	8017	8039	8053	8059
8069	8081	8087	8089	8093	8101	8111	8117	8123	8147	8161	8167	8171
8179	8191	8209	8219	8221	8231	8233	8237	8243	8263	8269	8273	8287
8291	8293	8297	8311	8317	8329	8353	8363	8369	8377	8387	8389	8419
8423	8429	8431	8443	8447	8461	8467	8501	8513	8521	8527	8537	8539
8543	8563	8573	8581	8597	8599	8609	8623	8627	8629	8641	8647	8663
8669	8677	8681	8689	8693	8699	8707	8713	8719	8731	8737	8741	8747
8753	8761	8779	8783	8803	8807	8819	8821	8831	8837	8839	8849	8861

```
8863  8867  8887  8893  8923  8929  8933  8941  8951  8963  8969  8971  8999
9001  9007  9011  9013  9029  9041  9043  9049  9059  9067  9091  9103  9109
9127  9133  9137  9151  9157  9161  9173  9181  9187  9199  9203  9209  9221
9227  9239  9241  9257  9277  9281  9283  9293  9311  9319  9323  9337  9341
9343  9349  9371  9377  9391  9397  9403  9413  9419  9421  9431  9433  9437
9439  9461  9463  9467  9473  9479  9491  9497  9511  9521  9533  9539  9547
9551  9587  9601  9613  9619  9623  9629  9631  9643  9649  9661  9677  9679
9689  9697  9719  9721  9733  9739  9743  9749  9767  9769  9781  9787  9791
9803  9811  9817  9829  9833  9839  9851  9857  9859  9871  9883  9887  9901
9907  9923  9929  9931  9941  9949  9967  9973
```

A-5 Pentru exercițiul 6.5.5:

Exemplu de fişier cu date “T55.DAT”:

```
Iliescu Dan                  : 5.89
Gheorghe Adrian              :10.00
Alexandrescu Valentin        : 8.85
Zorila Sorin Claudiu         : 7.26
Achim Horatiu                : 8.88
Pascu Cristina               : 9.83
Crisan Adelina               : 9.57
Olteanu Cristian             : 7.22
```

Conținutul fişierului “T55.REZ” cu rezultate:

```
       Elevii ordonati in ordinea mediilor :


  1. Gheorghe Adrian               Media: 10.00
  2. Pascu Cristina                Media:  9.83
  3. Crisan Adelina                Media:  9.57
  4. Achim Horatiu                 Media:  8.88
  5. Alexandrescu Valentin         Media:  8.85
  6. Zorila Sorin Claudiu          Media:  7.26
  7. Olteanu Cristian              Media:  7.22
  8. Iliescu Dan                   Media:  5.89
```

BIBLIOGRAFIE

1. COMPAQ FORTRAN. Language Reference Manual, Compaq Computer Corporation, Houston, Texas, 1999.
2. E. W. Dijkstra – A Discipline of Programming, Prentice-Hall, Englewood Cliffs, New Jersey, 1976.
3. FORTRAN -77, Operare, I.T.C.I. Filiala Cluj-Napoca, 1988.
4. FORTRAN -77, Programare, I.T.C.I. Filiala Cluj-Napoca, 1988.
5. F. Gobesz, A. Cătărig – Programarea calculatoarelor electronice, Institutul Politehnic Cluj, 1974.
6. IBM FORTRAN/2, Language Reference, I.B.M. Corp., 1987
7. H. M. Kienle – History, Architecture & Evolution of Compilers, SENG420, University of Victoria, Canada, 2002.
8. A. C. Marshall – Fortran 90 Course Notes, The University of Liverpool, 1997.
9. S. Niculescu – FORTRAN, Inițiere în programare structurată, Editura Tehnică, Bucureşti, 1979.
10. E. Petac, C. Partale – Limbajul FORTRAN, Programare şi aplicații, Matrix Rom, Bucureşti, 2002.
11. M. Petrina, A. Cătărig – Programare, teorie şi aplicații pe PC-uri compatibile IBM, Universitatea Tehnică din Cluj-Napoca, 1993.
12. T. Roberts, S. Egdorf – A Face-Lift for Aging FORTRAN Scientific Applications, LA-UR 01-6629, Los Alamos National Laboratory, 2001.
13. G. E. Urroz – Getting started with GNU FORTRAN G77, Utah State University, 2002.
14. http://www.cfm.brown.edu/tutorials/Fortran.html
15. http://www.engineering.usu.edu/cee/faculty/gurro/
16. http://www.forceproject.hpg.ig.com.br/
17. http://www.fortran.com/
18. http://www.geocities.com/Athens/Olympus/5564/
19. http://www.geog.nottingham.ac.uk/~mather/useful/Computing.html#FORTRAN
20. http://www.hep.man.ac.uk/guide/unix_tutorial/section3.6.html
21. http://www.hpctec.mcc.ac.uk/hpctec/courses/Fortran90/mc_f90.html
22. http://www.imamod.ru/~vab/vfort/
23. http://www.levenez.com/lang/

Cartea prezintă noţiunile necesare abordării limbajului de programare Fortran pentru cei începători în domeniul realizării aplicaţiilor informatice. După o prezentare nu foarte exhaustivă a conceptelor generale referitoare la sisteme de calcul şi a unor instrumente pentru descrierea procedurilor, cititorul este introdus în limbajul de programare Fortran 77. Deşi acest limbaj este considerat demodat, permite totuşi însuşirea unor noţiuni şi tehnici simple, fundamentale pentru realizarea aplicaţiilor de tip consolă, fără a necesita un efort deosebit din partea utilizatorilor. Se prezintă şi aspectele esenţiale ale diferenţelor apărute în urma evoluţiei acestui limbaj de programare, iniţiind cititorul în Fortran 90 în cadrul unui capitol presărat cu exemple. Această versiune dispune deja de caracteristici ce-l recomandă ca fiind un excelent instrument în programarea legată de calcule ştiinţifice şi inginereşti, putând concura cu succes în acest domeniu cu limbajul C. După prezentarea scurtă a celui mai popular compilator necomercial al momentului (GNU Fortran 77), cititorului i se oferă un capitol întreg cu o serie de exerciţii şi soluţii propuse, cu un caracter didactic, în scopul dezvoltării experienţei practice şi a posibilităţii de a-şi testa cunoştinţele în acest domeniu.

Despre autori:

Zsongor F. GOBESZ este şef de lucrări la Universitatea Tehnică din Cluj-Napoca, Facultatea de Construcţii, Catedra de Mecanica Construcţiilor, susţinând cursuri în cadrul disciplinelor: programarea calculatoarelor, proiectare asistată de calculator, şi informatică în construcţii. Este absolvent al Institutului Politehnic din Cluj-Napoca (şi al unui curs postuniversitar de formare în domeniul analizei şi proiectării sistemelor informatice de gestiune pe minicalculatoare), a obţinut titlul ştiinţific de doctor cu o teză din domeniul sistemelor expert. A participat la numeroase programe naţionale şi internaţionale de cercetare, publicând peste 30 de lucrări ştiinţifice, în calitate de autor şi coautor. Este membru al Societăţii Maghiare Tehnico-Ştiinţifice din Transilvania, precum şi al corporaţiei doctorilor din Academia Maghiară de Ştiinţă. Domeniile sale de interes ştiinţific cuprind pe lângă programarea calculatoarelor, informatică aplicată, şi alte domenii cum ar fi: modelarea şi proiectarea asistate de calculator, statica construcţiilor şi analiza structurală.

Ciprian BACOŢIU este şef de lucrări la Universitatea Tehnică din Cluj-Napoca, Facultatea de Construcţii, Catedra de Instalaţii, susţinând cursuri în cadrul disciplinelor: alimentări cu apă, instalaţii sanitare, de canalizare şi gaze, hidraulică, precum şi echilibrarea hidraulică în tehnica instalaţiilor pentru construcţii. Este absolvent al Universităţii Tehnice din Cluj-Napoca şi a obţinut titlul de M.Sc. în ingineria mediului la Ecole Polytechnique Fédérale din Lausanne. Teza lui de doctorat tratează probleme legate de gestionarea sistemelor de canalizare. A participat la mai multe programe naţionale de cercetare şi este membru al Asociaţiei Inginerilor de Instalaţii. Domeniile sale de interes ştiinţific includ proiectarea instalaţiilor, programarea calculatoarelor, aplicaţiile informatice inginereşti, metodele multicriteriale, precum şi sistemele de informaţii geografice.